Gears4Net - Ein Programmiermodell für asynchrone Kommunikation in P2P-Systemen

Von der Fakultät für Ingenieurswissenschaften
der Universität Duisburg-Essen
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
genehmigte Dissertation

von
Martin Saternus
aus Bochum

Referent: Prof. Dr.-Ing. Torben Weis
Korreferent: Prof. Dr. phil. nat. Christian Becker
Tag der mündlichen Prüfung: 26.11.2010

für Günter

Danksagung

Ich möchte diese Gelegenheit nutzen, um mich bei all denjenigen zu bedanken, die mich während der Entstehung dieser Arbeit tatkräftig begleitet und unterstützt haben.

Mein besonderer Dank gilt meinen Gutachtern Torben und Christian. Danke Dir, Torben, für die weit über die akademische Arbeit hinausgehende Unterstützung und die freundschaftliche Betreuung. Dir, Christian, möchte ich besonders für die Einführung in die lebendige wissenschaftliche Diskussion während der gemeinsamen Zeit an der Universität Stuttgart danken.

Besonderer Dank gilt auch meinem langjährigen Zimmernachbarn und Freund Mirko. Danke, dass Du mir nicht nur in vielen akademischen Fragen sondern vor allem als Freund in guten und schlechten Zeiten beiseite gestanden hast. Weiterhin danke ich meinen Kollegen aus dem VS-Team Arno, Bernd, Matthäus, Matthias, Sebastian und Sebastian sowie Leonie und Kathrin und den weiteren Kolleginnen und Kollegen der CAR-Abteilung.

Einen weiteren großen Anteil an dieser Arbeit haben meine langjährigen Freunde, die mich jeder Zeit in vielfältiger Weise unterstützt haben. Danke an Britta, Nathalie, Patrick, Richard, Tim und Ute. Besonderer Dank gilt Maike. Danke, dass Du dieser Arbeit viel Zeit und deutlich mehr als nur ein Komma gespendet hast.

Last but not least möchte ich mich bei meiner Familie für die permanente Unterstützung und den richtigen Anstoß zur rechten Zeit bedanken. Danke Ulli, danke Kurt, danke Katrin. Weiterhin möchte ich mich noch bei Günter und Gerda bedanken. Danke, dass Ihr immer an mich geglaubt habt. Danke Maryla.

Martin Saternus
Mülheim an der Ruhr, Dezember 2010

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Entwicklung massiv paralleler, verteilter, ortsbezogener Anwendungen 3
1.2 Wissenschaftlicher Beitrag 7
1.3 Organisation der Arbeit 8

2 Grundlagen **11**

2.1 Generische Datentypen 11
2.2 Operatoren 13
2.3 Deleganten 16
2.4 Zeitgeber 18
2.5 Thread Synchronisierung mit graphischen Steuerelementen 21
2.6 Iteratoren 23

3 Gears4Net **29**

3.1 Motivation 29
3.2 Programmiermodellanalyse 30
3.2.1 Modell I: Blockierende Programmierung 31
3.2.2 Modell II: Nicht-blockierende Programmierung 36
3.2.3 Modell III: Asynchrone Programmierung 38
3.2.4 Modell IV: Aktor-basierte Programmierung 40
3.3 Anforderungsanalyse 41
3.4 Das Gears4Net Programmiermodell 42
3.4.1 Grundidee des Programmiermodells Gears4Net 43
3.4.2 Die Gears4Net Modellarchitektur 45
3.5 Plattform- und Sprachwahl 51
3.6 Implementierung des Gears4Net Frameworks 52
3.6.1 Message-Queues 53
3.6.2 Signals 56
3.6.3 Receiver 57
3.6.4 State-Machine 82

3.6.5 Die ProtocolBase-Klasse . . . 94
3.6.6 Scheduler . . . 103
3.7 Anwendungsszenarien und Beispiele . . . 113
3.7.1 Grundlagen der Netzwerkprogrammierung . . . 113
3.7.2 Drei-Wege-Handshake . . . 116
3.7.3 Existenz- und Erreichbarkeitsverfahren . . . 118
3.8 Evaluation . . . 122
3.8.1 Ressourceneffizienz . . . 122
3.8.2 Quellcodestrukturierung . . . 128
3.8.3 Synchronisierung . . . 129
3.8.4 Ausführungsumgebung der Protocol-Instanzen . . . 129
3.9 Weiterentwicklung des Programmiermodells Gears4Net . . . 133
3.9.1 Unveränderbare Nachrichten . . . 134
3.9.2 Restriktion auf Nachrichten-basierte Kommunikation . . . 138
3.9.3 Verhinderung blockierender Systemaufrufe . . . 141
3.10 Verwandte Arbeiten . . . 141
3.10.1 Comega und Polyphonic C# . . . 141
3.10.2 Concurrency and Coordination Runtime . . . 144
3.10.3 Parallel Extensions . . . 147
3.10.4 Parallel C# . . . 148
3.10.5 Scala . . . 149
3.10.6 SEDA . . . 153
3.10.7 Pfadausdrücke . . . 155
3.10.8 Monitore . . . 156
3.11 Zusammenfassung . . . 157

4 Symstry **159**
4.1 Motivation . . . 159
4.2 System-Modell und Anforderungsanalyse . . . 160
4.3 Symbolische Adressen . . . 162
4.4 Operationen des Symstry-Protokolls . . . 163
4.4.1 Routing . . . 163
4.4.2 Eintrittsprotokoll . . . 170
4.4.3 Austrittsprotokoll . . . 172
4.4.4 Selbststabilisierung . . . 172
4.4.5 GeoCast . . . 173
4.5 Implementierung des Symstry-Protokolls . . . 175
4.6 Anwendungsszenario . . . 178

4.7 Evaluation 180
4.8 Weiterentwicklung und weiterer Forschungsbedarf 184
4.9 Verwandte Arbeiten 185
4.9.1 Pastry 185
4.9.2 Geostry 186
4.9.3 GeoCast Verfahren 187
4.10 Zusammenfassung 189

5 Peers@Play-Inspector 191
5.1 Motivation 191
5.2 Architektur von P2P-basierten Anwendungen 192
5.3 Anforderungsanalyse 194
5.4 Die Simulationsumgebung Peers@Play-Inspector 195
5.4.1 Netzwerkspezifikation 196
5.4.2 Netzwerk-, Analyse- und Instanzmanagement 200
5.4.3 Nachrichtenverwaltung 204
5.5 Zusammenfassung und Ausblick 205

6 Fazit 207

Abkürzungsverzeichnis 211

Literaturverzeichnis 213

Abbildungsverzeichnis

1.1 Betrachtete Forschungsgebiete 3
1.2 Einordnung der Themenschwerpunkte in die Gesamtarchitektur 4

2.1 Deklaration generischen Klassen, Structs und Interfaces 13
2.2 Iteratordeklaration 24
2.3 Diagramm der IEnumerator-Schnittstelle 27

3.1 Thread Erzeugnung - Windows 7 - 32 Bit - Stack: 512 KB 33
3.2 Thread Erzeugnung - Windows 7 - 32 Bit - Stack: 256 KB 33
3.3 Thread Erzeugnung - Windows 2008 - 64 Bit - Stack: 512 KB 34
3.4 Übersicht des Gears4Net Architekturdiagramms 46
3.5 Klassendiagramm der AbstractMessageQueue-Basisklasse 53
3.6 Klassendiagramm der Signal-Klasse 56
3.7 Klassendiagramm der ReceiverBase-Klasse 60
3.8 Darstellung der Wartebedingung *((a | b) & c)* 62
3.9 Klassendiagramm der ParallelReceiver-Klasse 75
3.10 Sequenzdiagramm eines asynchronen Methodenaufrufs 79
3.11 Asynchroner Methodenaufruf mittels eines AsyncResultReceivers . . . 80
3.12 Klassendiagramm der AsyncResultReceiver-Klasse 82
3.13 Klassendiagramm der AbstractStateMachine-Klasse 88
3.14 Klassendiagramm der StateMachine-Klasse mit drei generischen Parametern 93
3.15 Klassendiagramm der ProtocolBase-Klasse 95
3.16 Klassendiagramm der Scheduler-Klasse 104
3.17 Zustandsautomat zum Aufbau einer Netzwerkverbindung und Ausführung der Applikationslogik 114
3.18 Zustandsautomat des Drei-Wege-Handshake 117
3.19 Speicherverbrauch bei der Erzeugung von Iteratoren 124
3.20 Virtueller Speicherverbrauch bei der Erstellung von Protocol-Instanzen 126
3.21 Speicherverbrauch bei der Erstellung von Protocol-Instanzen 126
3.22 Speicherverbrauch pro Protocol-Instanz 127

3.23 Nachrichtenverarbeitung pro Sekunde in Abhängigkeit von der Anzahl der Protocol-Instanzen pro Scheduler 132
3.24 Nachrichtenverarbeitung pro Sekunde in Abhängigkeit von der Anzahl der Scheduler sowie der Gesamtanzahl der Protocol-Instanzen . . 132

4.1 Geographische Adressen 162
4.2 Organisatorische Adressen 162
4.3 Symstry Routing-Baum 165
4.4 Suche nach dem nächstgelegenen Knoten mithilfe des Rekursionsverfahrens 168
4.5 Eintrittsprotokoll des Symstry-Netzes 171
4.6 Versand einer GeoCast-Nachricht 174
4.7 Lokationsbezogener Messenger 180
4.8 Anzahl der Einträge des durchschnittlichen Routing-Baumes in Abhängigkeit der Netzwerkalterungssimulation 182
4.9 Anzahl der Routingschritte in Abhängigkeit der Netzwerkalterungssimulation 183
4.10 Tiefe der GeoCast-Spannbaumes in Abhängigkeit der Netzwerkalterungssimulation und der prozentualen Größe des Zielgebietes 184

5.1 Schichtenarchitektur einer P2P-Instanz nach Aberer 192
5.2 Schichtenarchitektur einer P2P-Instanz nach Weis 193
5.3 Peers@Play-Inspector 196
5.4 Netzwerkspezifikationsbereich des Peers@Play-Inspectors 197
5.5 Abbildung der Inspector-Konfiguration auf das Schichtenmodell 198
5.6 Darstellung des Instance-Explorers mit zugehörigem Kontextmenü . . 201
5.7 Registerkarte Assembly-Loader 203
5.8 Darstellung der eingehenden Nachrichten im Message- und Details-View 204

Auflistungsverzeichnis

2.1 ArrayList vs. generische Liste des Typs *Integer* 12
2.2 Operatoren- vs. Funktionsaufrufdarstellung in C# 15
2.3 Gleich- und Ungleich-Operator 16
2.4 Deklaration eines typsicheren Funktionszeigers 16
2.5 Intermediate Language Code des in 2.4 definierten Deleganten 17
2.6 System.Threading.Timer 19
2.7 System.Timers.Timer 19
2.8 System.Windows.Forms.Timer 20
2.9 Invoke am Beispiel der Klasse System.Windows.Forms.Form 22
2.10 Verschachtelte *foreach*-Schleife 23
2.11 Implizite und explizite Verwendung eines Iterators 25
2.12 Ausschnitt aus der generierten Enumeratorklasse 28

3.1 Polling auf zwei Sockets 37
3.2 Asynchroner Zugriff aus zwei Socket-Instanzen 39
3.3 Unterbrechbare Methoden und Wartebedingungen 44
3.4 Generische DequeueMessage Methode der Klasse AbstractMessage-Queue 55
3.5 Auszug der überladenen Operatoren der ReceiverBase-Klasse 64
3.6 Überladener PLUS-Operator der Klasse ReceiverBase<T> 65
3.7 Signatur der Funktionszeiger ReceiveHandler und Filter 66
3.8 Nachrichtenverarbeitung in der HandleMessage-Methode 67
3.9 Methoden der Verwendung des OrReceivers 71
3.10 Generischer Funktionszeiger StateMachineStarter 72
3.11 HandleMessage-Methode des PersistentReceivers 74
3.12 Verwendung eines ParallelReceivers 74
3.13 Beispielhafte Nutzung des IntervalReceivers 78
3.14 Verwendung des AsyncResultReceivers 81
3.15 Implementierung der Fibonacci Funktion in einem Iterator 84
3.16 Expliziter Aufruf des Fibonacci Iterators 85

3.17 Auszug aus der generierten Fibonacci Iterator Klasse 86
3.18 Iterator einer AbstractStateMachine mit drei Parametern 87
3.19 Nachrichtenverarbeitungsschritte 89
3.20 Nachrichtenzustellung an einen Receiver innerhalb der Run-Methode 90
3.21 HandleMessage-Methode 91
3.22 CreateEnumerator- und Clone-Methode der StateMachine-Klasse . . . 94
3.23 Erstellung einer StateMachine 94
3.24 Implementierung der Run-Methode der ProtocolBase-Klasse 99
3.25 Vergleich der Darstellungsformen von Wartebedingungen 101
3.26 Implementierung des Launch-Kommandos mit 2 generischen Parametern 102
3.27 Schedulingverfahren des STASchedulers 105
3.28 Wakeup-Methode des STASchedulers 106
3.29 Implementierung der WinFormsSchedulerHelperControl-Klasse 110
3.30 Schedulingverfahren des WinFormsScheduler 112
3.31 Gears4Net Implementierung zum Aufbau einer Netzwerkverbindung und Ausführung der Applikationslogik 115
3.32 Implementierung des Drei-Wege-Handshakes 119
3.33 Implementierung des Ping-Pong Verfahrens 121
3.34 Strukturierung des Quelltextes im Gears4Net-Programmiermodell . . 128
3.35 Protocol zum Test des Nachrichtendurchsatzes 130
3.36 Gears4Net Standardnachricht 134
3.37 Gears4Net Nachricht mit dem Schlüsselwort message 135
3.38 Implementierung des Copy-On-Write Konzepts 137
3.39 Verwendung des Schlüsselwortes protocol 139
3.40 Generierter protocol Quellcode 140
3.41 Asynchroner Methodenaufruf 142
3.42 Choice Arbiter 146
3.43 Join Arbiter 146
3.44 Matrixmultiplikation mit Parallel.For 147
3.45 Ping-Pong Szenario auf Basis des Aktorenmodells der Sprache Scala . 151

4.1 Rekursives Verfahren zur Bestimmung des nächstgelegenen Knotens . 167
4.2 Auszug aus der Implementierung des Symstry-Protokolls 176

Tabellenverzeichnis

2.1 Einschränkungen von Typparametern 13
2.2 Operatoren der Sprache C# 14
2.3 Überladbare Operatoren 15
2.4 Vergleich der drei Zeitgeberkonzepte 20

3.1 Zusammenfassung der Ergebnisse bei der Erzeugung maximal vieler Threads in unterschiedlichen Konfigurationen 35
3.2 Zusammenfassung der Programmiermodellanalyse - Die Angaben des Aktorenmodells basieren auf dem jeweilig optimalen Programmiermodell für den betrachteten Analyseaspekt. 42
3.3 Entwicklungsplattform- und Spracheigenschaften 52
3.4 Kurzbeschreibung der Receiver des Gears4Net Frameworks 58
3.5 Abbildung der Operatoren auf spezialisierte Receiver 63

Kapitel 1

Einleitung

Die Entwicklung und Wartung skalierender *Client-Server*-Anwendungen, die die Integrität und Konsistenz der Daten sowie die permanente Erreichbarkeit des Systems unter nahezu beliebiger Last garantieren, ist extrem kostenintensiv. *Peer-to-Peer*-basierte Ansätze bilden im Gegensatz dazu eine kostengünstige Alternative, da diese die Rechenleistung und Datenhaltung des Gesamtsystems auf alle Teilnehmer des Systems verteilen und somit auf den Einsatz kostenintensiver Serverinfrastrukturen verzichten können.

Die wachsende Attraktivität *Peer-to-Peer*-basierter Anwendungen und der damit verbundene Paradigmenwechsel in der Architektur großer, skalierender Systeme wird von zwei Entwicklungsschwerpunkten forciert. Der erste Schwerpunkt betrifft die Entwicklungen bei der Prozessortechnologie, der zweite die Verbreitung von Breitbandnetzwerken. Die zunehmende Ausbreitung von *Multi-Core*-Prozessoren in Geräten des Endkundensegments, wie beispielsweise Heim-PCs oder Notebooks, führt zu einer enormen Steigerung der Rechenleistung und erlaubt erstmals ein paralleles Anwendungsdesign außerhalb kostspieliger Serverumgebungen. Der zweite die *Peer-to-Peer*-Technologie begünstigende Aspekt ist die zunehmende Verbreitung von stationären und mobilen Breitbandnetzen, die eine schnelle Kommunikation zwischen den Teilnehmern des Systems ermöglicht.

Die Anwendungsgebiete *Peer-to-Peer*-basierter Lösungen sind vielfältig. Die unterschiedlichen Optimierungsziele der einzelnen Systemvarianten ermöglichen die Substitution Server-basierter Systeme in verschiedensten Anwendungsdomänen. Ein mögliches Anwendungsfeld bilden ortsbezogene Anwendungen, die heutzutage größtenteils auf Basis klassischer *Client-Server*-Architekturen entwickelt werden. Anwendungen dieser Klasse, wie beispielsweise das *Google - Places Directory* [42], stellen Informationen und Daten über die lokale Umgebung eines Benutzers beziehungsweise zu ausgewählten Orten oder Regionen bereit.

Der Softwareentwicklungsprozess *Peer-to-Peer*-basierter Anwendungen und Dienste, der in dieser Arbeit am Beispiel einer ortsbezogenen Anwendung vorgestellt wird, gestaltet sich jedoch im Vergleich zu klassischen Server-basierten Systemen deutlich komplexer. Die gesteigerte Komplexität resultiert aus der massiven Parallelität des Gesamtsystems und der damit einhergehenden Kommunikation der einzelnen Systemkomponenten. Zudem gestalten sich die Test- beziehungsweise Simulationsmethoden einer verteilten Lösung deutlich komplexer, als dies bei einem zentralisierten Ansatz der Fall ist.

Die vorliegende Arbeit, die die Entwicklungsmethoden *Peer-to-Peer*-basierter Systeme am Beispiel eines ortsbezogenen Anwendungsszenarios betrachtet, wirft neue Fragestellungen auf und stellt unterschiedliche Forschungsbereiche vor neue Herausforderungen. Die drei wesentlich tangierten Forschungsgebiete werden in Abbildung 1.1 vorgestellt und anschließend beschrieben.

Der erste Forschungsbereich, der sich den Herausforderungen durch die Entwicklung *Peer-to-Peer*-basierter Systeme stellen muss, ist der Bereich des Software Engineering. Die angesprochene und rasant zunehmende Verbreitung von *Multi-Core*-Prozessoren forciert die Architektur echt paralleler Anwendungen, bei denen einzelne Komponenten auf unterschiedlichen Prozessorkernen ausgeführt werden. Die Beherrschung des zunehmenden Parallelisierungsgrades ist mit den aus den Anfängen der Betriebssystementwicklung stammenden Synchronisierungsverfahren nahezu unmöglich, da diese ausschließlich feingranulare Verfahren bereitstellen und Parallelität nicht als Anwendungsdesignkriterium betrachten. Die Komplexität des Einsatzes feingranularer Synchronisierungsverfahren steigt überproportional mit dem Parallelisierungsgrad und bietet damit den optimalen Nährboden für *Race-Conditions* und *Dead-Locks*. Die Konzeption neuer Softwareentwicklungsmethoden zum Design, zur Entwicklung und zum Test beziehungsweise zur Simulation massiv paralleler Systeme ist somit notwendig.

Die Entwicklung *Peer-to-Peer*-basierter Systeme entspringt dem zweiten Forschungsbereich, den Verteilten Systemen. Der Forschungsschwerpunkt dieses Bereiches liegt beispielsweise in der Konzeption geeigneter Verfahren zum Aufbau und Betrieb von *Peer-to-Peer*-Systemen. Zudem liegen die Betrachtung der dem System zugrundeliegenden Kommunikationsinfrastruktur sowie die Konzeption von effizienten *Routing*-Verfahren im Fokus dieses Forschungsbereiches.

Der dritte und letzte Forschungsbereich, der in Abbildung 1.1 dargestellt ist, betrifft lokationsbezogenen Applikationen. Die Bestimmung beziehungsweise Spezifikation eines Ortes oder einer Region ist jedoch abhängig von der Anwendungsdomäne nicht

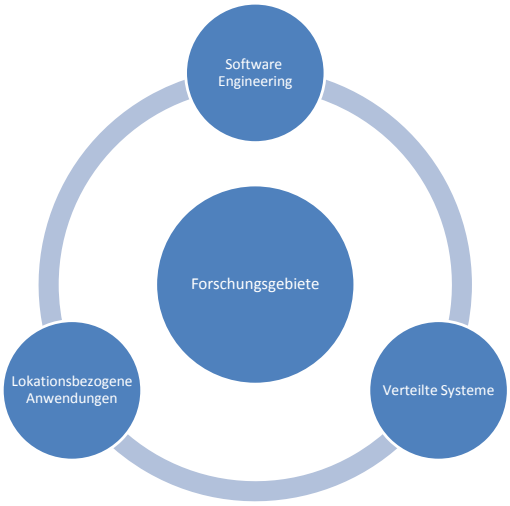

Abbildung 1.1: Betrachtete Forschungsgebiete

trivial. Klassische Positionierungssysteme wie das weit verbreitete *Global Positioning System (GPS)* versagen beispielsweise bei der Verwendung innerhalb von Gebäuden oder bei der intuitiven, menschenverständlichen Darstellung eines Ortes. Ein möglicher Ansatz zur Vereinfachung der Spezifikation von Orten und Regionen besteht in der Einführung von Systemen, die auf symbolischen Koordinaten und nicht auf geometrischen Koordinaten mit Längen- und Breitengrad sowie der Höhe des Ortes operieren. Symbolische Adressen und Koordinaten arbeiten unabhängig von den Längen- und Breitengraden und erinnern in ihrer Struktur an postalische Adressen. Die Herausforderungen im dritten Forschungsgebiet konzentrieren sich daher auf die Entwicklung eines *Peer-to-Peer*-Systems, das Orte und Regionen mit intuitiv verständlichen symbolischen Koordinaten adressieren kann und somit die Grundlage für weitergehende ortsbezogene Anwendungen bietet.

1.1 Entwicklung massiv paralleler, verteilter, ortsbezogener Anwendungen

Die Entwicklung massiv paralleler, verteilter, ortsbezogener Anwendungen stellt jedes der drei vorgestellten Forschungsgebiete vor neue Fragestellungen und Herausforderungen. Die aufgezeigten Beiträge, die jeden einzelnen der beiden Bereiche des Software Engineerings, der Verteilten Systeme sowie den Forschungsbereich der lokationsbezogenen Anwendungen einbringen, um die Gesamtkomplexität des Entwicklungsprozesses zu reduzieren, müssen bereichsübergreifend betrachtet werden.

Der erste Themenschwerpunkt, der sich vorzugsweise der Methoden und Verfahren der Forschungsgebiete des Software Engineerings und der Verteilten Systeme bedient, behandelt die Entwicklung geeigneter Programmier- und Designmodelle für parallel arbeitende Anwendungen und Systeme. Der zweite Forschungsschwerpunkt betrachtet die Entwicklung eines auf symbolischen Koordinaten basierenden *Peer-to-Peer*-Systems. Dieser Themenschwerpunkt bildet die architektonische Grundlage für die Entwicklung von verteilten, ortsbezogenen Anwendungen und beinhaltet Methoden aller drei Forschungsbereiche. Die Bereitstellung von Test- und Simulationswerkzeugen und -verfahren charakterisiert den dritten und letzten Themenschwerpunkt, der sich der Konzepte des Software Engineerings sowie der verteilten Systeme bedient.

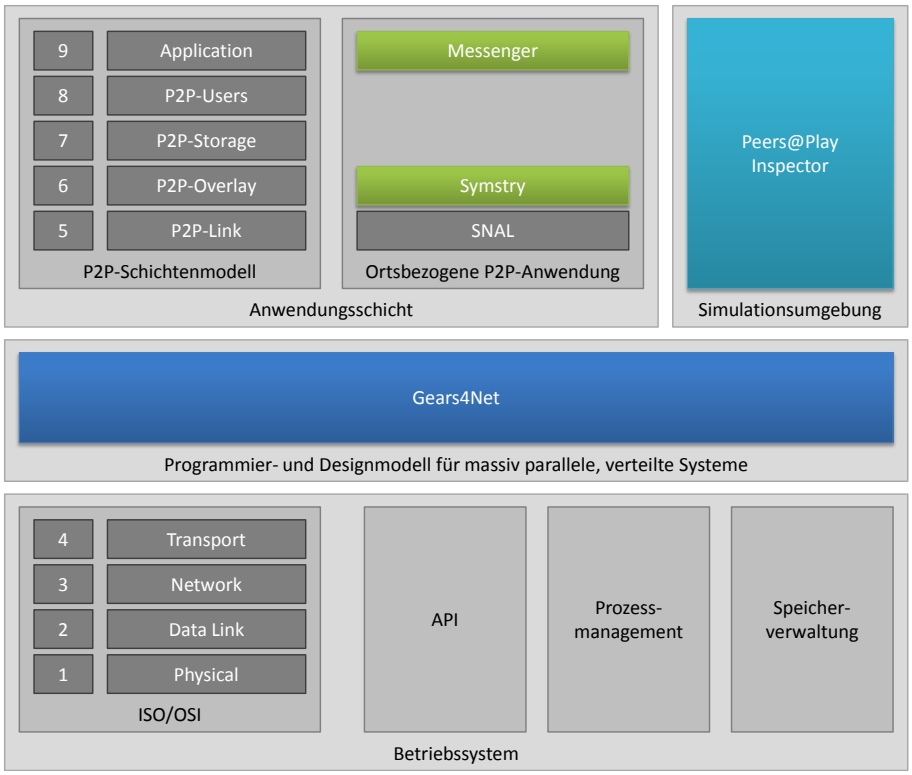

Abbildung 1.2: Einordnung der Themenschwerpunkte in die Gesamtarchitektur

Die drei Themenschwerpunkte werden als autarke Bausteine betrachtet werden, die als Bestandteile eines Schichtenmodells zu einer Gesamtarchitektur verschmelzen. Die Einordnung der einzelnen Bausteine in den Gesamtkontext erfolgt im Architekturdiagramm in Abbildung 1.2. Jeder der drei Bausteine wird im Folgenden beschrieben und ist farblich markiert und mit einem Projektnamen versehen als Teil des Schichtenmodells dargestellt.

Gears4Net

Der zunehmende Parallelisierungsgrad in der Anwendungsentwicklung wird durch die zwei Aspekte der Verbreitung Verteilter Systeme und die Zunahme von *Multi-Core*-Prozessorarchitekturen forciert. Der Entwurf eines Programmiermodells für diese Anwendungsklasse ist somit eine direkte Reaktion auf die veränderten Rahmenbedingungen im Softwareentwicklungsprozess. Das neue Programmier- und Designmodell, das aufbauend auf der Schicht des Betriebssystems konzipiert wird, trägt den Projektnamen *Gears4Net* und ist im Architekturdiagramm in Abbildung 1.2 blau markiert. *Gears4Net* bietet die nötige Unabhängigkeit von der zugrundeliegenden Betriebssysteminfrastruktur und ermöglicht gleichzeitig die einfache Nutzung des Modells durch die im Architekturdiagramm oberhalb angesiedelten Schichten.

Die Zielsetzung, die der Entwicklung des *Gears4Net*-Programmiermodells zugrundeliegt, ist die Reduzierung der Entwicklungskomplexität, die durch den Einsatz von parallel arbeitenden *Threads* und den daraus entstehenden Synchronisations- und Nebenläufigkeitseffekten entsteht. Das *Gears4Net*-Programmiermodell vereint dazu die Vorteile der effizienten Ressourcennutzung der asynchronen Programmierung [113, 119] mit der Quellcodestrukturierung des blockierenden Programmiermodells [14, 19, 113, 119] und paart diese mit dem von Hewitt, Bishop und Steiger vorgestellten Aktorenmodell [3, 4, 25]. Das Aktorenmodell fordert den Einsatz von autarken als Aktoren bezeichneten Einheiten, die in sich sequentiell arbeiten und mit der Außenwelt, beispielsweise parallel ausgeführten Aktoren, über asynchronen Nachrichtenaustausch kommunizieren. Die Eigenschaft der sequentiellen Verarbeitung innerhalb eines Aktors ermöglicht den Verzicht auf etwaige Synchronisationsmechanismen und führt damit zu einer signifikanten Reduktion der Entwicklungskomplexität.

Symstry

Die Entwicklung eines grundlegenden, auf symbolischen Koordinaten basierenden *Peer-to-Peer*-Systems, das von aufbauenden ortsbezogenen Anwendungen genutzt werden kann, ist Gegenstand des zweiten Themenschwerpunktes. Der Baustein mit dem Projektnamen *Symstry*, der als Teil der Anwendungsschicht konzipiert wurde, ist im Architekturdiagramm in Abbildung 1.2 grün markiert oberhalb des *Gears4Net*-Programmiermodells angesiedelt. Der Themenschwerpunkt enthält neben der Konzeption und Entwicklung des Grundsystems noch den ebenfalls in der Farbe Grün markierten Anwendungsfall, den lokationsbezogenen *Messenger*.

Die Struktur des auf symbolischen Koordinaten basierenden *Peer-to-Peer*-Systems und der darauf aufbauenden Messenger-Anwendungen folgen dem von Weis [126]

vorgestellten *Peer-to-Peer*-Schichtenmodell. Letzteres setzt auf der vierten Schicht des ISO/OSI-Modells auf und klassifiziert die unterschiedlichen Schichten einer *Peer-to-Peer*-Architektur. Das im Rahmen dieser Arbeit vorgestellte ortsbezogene *Peer-to-Peer*-System *Symstry* ist auf Schicht sechs des Modell, dem *P2P-Overlay*, die Anwendung *Messenger* auf Schicht neun, dem *Application-Layer*, angesiedelt.

Sowohl das *Symstry-Peer-to-Peer*-System als auch der Anwendungsfall *Messenger* nutzen das Programmier- und Designmodell *Gears4Net*, da die erforderlichen Netzwerkprotokolle durch ein massiv paralleles Verhalten gekennzeichnet sind. Die unterliegende *Gears4Net*-Schicht trägt somit signifikant zur Reduktion der Entwicklungskomplexität bei und ermöglicht die Konzentration auf die im Protokoll enthaltenen Verfahren und nicht auf die durch die Parallelität entstehenden Problemstellungen.

Peers@Play Inspector

Der dritte der betrachteten Themenschwerpunkte beleuchtet die Entwicklung einer geeigneten Test- und Simulationsumgebung für *Peer-to-Peer*-basierte Systeme. Der in Türkis dargestellte Baustein mit dem Projektnamen *Peers@Play-Inspector* wurde aufbauend auf dem *Gears4Net*-Programmiermodell und parallel zur *Peer-to-Peer*-basierten Anwendungsschicht konzipiert. Die Zielsetzung bei der Entwicklung der Test- und Simulationsumgebung besteht in der Bereitstellung eines geeignet großen Testbetts, der strukturellen Orientierung am vorgestellten *Peer-to-Peer*-Schichtenmodell sowie der Möglichkeit zur Simulation von produktiv eingesetztem Quellcode.

Die Herausforderungen an die drei Zielsetzungen sind dabei unterschiedlicher Natur. Bereitstellung eines geeignet großen Testbetts und die damit einhergehende Instanziierung einer Vielzahl an *Peer-to-Peer*-Knoten erfordert eine massiv parallel arbeitenden Simulationsumgebung. Die strukturelle Orientierung des *Peer-to-Peer*-Schichtenmodells erfordert zudem die Konfiguration einer Knoteninstanz, die sich aus verschiedenen Protokollen der einzelnen Schichten zusammensetzen kann. Die abschließende Forderung an die Test- und Simulationsumgebung bildet die bildet der Einsatz von Produktivquellcode zur Vermeidung von Divergenzen zwischen Produktiv- und Simulationscode.

1.2 Wissenschaftlicher Beitrag

Der wissenschaftliche Beitrag dieser Arbeit folgt direkt aus den drei im vorangegangenen Abschnitt beschriebenen Themenschwerpunkten. Die wesentlichen Beiträge der Projekte *Gears4Net*, *Symstry* und *Peers@Play-Inspector*, die in den Kapiteln 3-5 aufgezeigt werden, sind in den folgenden Auflistungen skizziert.

Gears4Net

- Entwurf eines geeigneten Programmier- und Designmodells zur Entwicklung massiv paralleler, verteilter Systeme auf Basis der asynchronen Programmierung und des Aktorenmodells.

- Realisierung des *Gears4Net*-Programmiermodells auf Basis der standardisierten Programmiersprache *C#* unter Ausnutzung des Iteratorenkonzeptes zur Implementierung unterbrechbarer Methoden.

Symstry

- Entwicklung eines P2P-Systems mit symbolischen Koordinaten als Grundlage für die Entwicklung von ortsbezogenen Anwendungen.

- Entwicklung eines effizienten *Routing*-Verfahrens, das Nachrichten innerhalb des *Overlay*-Netzwerkes in *O(log n)* Schritten zustellt.

- Konzeption eines effizienten *Geo-Cast*-Algorithmus, der einem Zielgebiet mit n Knoten eine Nachricht mittels eines Spannbaumes zustellt, wobei die Tiefe des Baumes *O(log n)* nicht überschreitet.

- Realisierung der Beispielanwendung *Messenger*, der auf Basis des *Symstry*-P2P-Systems ortsbezogene Gruppenchats errichtet.

Peers@Play-Inspector

- Anforderungsanalyse für eine Test- und Simulationsumgebung für P2P-basierte Systeme und Anwendungen.

- Entwicklung des Simulationswerkzeuges *Peers@Play-Inspector* zur Durchführung und zum Management von großen Simulationsläufen mit produktiv genutzten P2P-Komponenten.

1.3 Organisation der Arbeit

Der zentrale Fokus dieser Arbeit liegt auf dem Design und der Entwicklung eines Programmiermodells für parallele, verteilte Anwendungen. Darüber hinaus werden im Zuge dieser Arbeit das lokationsbezogene *P2P*-System *Symstry* und die Simulationsumgebung *Peers@Play-Inspector* vorgestellt, deren Entwicklung auf dem neuen Programmiermodell fußt. Die Arbeit ist dazu in sechs Kapitel aufgeteilt, die im Folgenden kurz vorgestellt werden.

Kapitel 2: Grundlagen
Im zweiten Kapitel werden die für die Implementierung des *Gears4Net*-Programmiermodells, des *Symstry*-P2P-Systems sowie der Simulationsumgebung relevanten Eigenheiten und Details der gewählten Programmierumgebung, des *Microsoft .NET Frameworks 3.5* und der Programmiersprache *C#* erläutert.

Kapitel 3: Gears4Net
Im dritten Kapitel werden die bestehenden Programmiermodelle für die Entwicklung von parallelen, verteilten Systemen analysiert und ihre Vor- beziehungsweise Nachteile beleuchtet. Im Anschluss an die Analysephase wird das neue Programmiermodell mit dem Namen *Gears4Net* vorgestellt, dessen Implementierung im weiteren Verlauf des Kapitels diskutiert wird. Abgeschlossen wird das dritte Kapitel mit der Evaluation des Programmiermodells inklusive der Präsentation von Beispielimplementierungen, die auf dem Programmiermodell beruhen.

Kapitel 4: Symstry
Im Vierten Kapitel wird das auf symbolischen Koordinaten basierende P2P-System *Symstry* vorgestellt, und zudem werden die Auswirkungen dieser Koordinaten auf *Routing*-Verfahren und *GeoCast*-Mechanismen näher erläutert. Den Abschluss des Kapitels bildet die Vorstellung der auf dem *Gears4Net*-Modell basierende Implementierung.

Kapitel 5: Peers@Play-Inspector
Das fünfte Kapitel beschreibt einen Simulator für P2P-Systeme, der auf Basis eines fünfstufigen Schichtenmodells [126] entwickelt wurde. Die in diesem Kapitel vorgestellte Simulationsumgebung, der *Peers@Play-Inspector*, bildet die Grundlage für die Evaluation des im vierten Kapitel vorgestellten P2P-Systems *Symstry*.

Kapitel 6: Fazit
Das Kapitel „Fazit“ enthält die Zusammenfassung der Ergebnisse dieser Arbeit und gibt zudem einen Ausblick auf mögliche Weiterentwicklungen und zusätzlichen Forschungsbedarf.

Kapitel 2

Grundlagen

Die folgenden Kapitel beschreiben das Programmiermodell *Gears4Net*, das darauf aufbauende P2P-System *Symstry* sowie die ebenfalls auf *Gears4Net* basierende Simulationsumgebung. Jedes der drei Projekte stellt neben der theoretischen Betrachtung eine reale Implementierung bereit, die in der ISO [6] und ECMA [38] standardisierten Sprache *C-Sharp (C#)* auf Basis des *Microsoft .NET Frameworks 3.5* vorgenommen wurde. Das folgende Grundlagenkapitel ist daher entwicklungstechnischer Natur und beschreibt die Besonderheiten und Details der objektorientierten Sprache *C#* sowie des *Microsoft .NET Frameworks*, die für das Verständnis der späteren Architektur hilfreich sind.

2.1 Generische Datentypen

Generische Datentypen [70], auch Generika genannt, sind seit der Version 2.0 fester Bestandteil des *Microsoft .NET Frameworks*. Die Einführung des Typparameterkonzeptes ermöglicht die Nutzung und Entwicklung von Datentypen, bei denen die Angabe eines oder mehrerer Typen bis zur Deklaration der Generika verzögert wird.

Die Verwendung parametrisierter Datentypen führt zu streng typisierten Ausdrücken, die durch den Compiler verifiziert werden können. Dies verhindert Laufzeitfehler und erhöht zudem die Ausführungsgeschwindigkeit des generierten Quellcodes. Auflistung 2.1 verdeutlicht dies am Beispiel zweier mit *Integer*-Instanzen gefüllten Listen, wobei die erste, in Zeile 1 deklarierte Liste des Typs *ArrayList* beliebige Objekte aufnehmen kann, während die zweite Liste in Zeile 5 die generische Variante des Typs *List<int>* instantiiert.

Die erste Liste, die die Aufnahme beliebiger Typen erlaubt, billigt somit die in den Zeilen 2 und 3 des Beispiels dargestellten Operationen und fügt den *Integer*-Wert 42 sowie das *String*-Datum in Zeile 3 der Liste hinzu. Das beschriebene Szenario bietet den optimalen Nährboden für die Entstehung von Laufzeitfehlern, die nicht automatisiert erkannt, sondern durch aufwändige Produkttests aufgefunden werden müssen. Solche Laufzeitfehler entstehen beispielsweise, sobald ein Entwickler in der Annahme über die Liste traversiert, dass diese ausschließlich Instanzen des Typs *Integer* enthalte. Im aufgezeigten Fall würde das Programm bei der *Cast*-Operation des *String*-Datums an Position *1* der Liste auf den Datentyp *Integer* eine *InvalidCastExeception* auslösen.

```
1    ArrayList standardList = new ArrayList();
2    standardList.Add(42);
3    standardList.Add("42");
4
5    List<int> genericList = new List<int>();
6    genericList.Add(42);
7    genericList.Add("42");      // Compile Time Error
```

Auflistung 2.1: ArrayList vs. generische Liste des Typs *Integer*

Um das skizzierte Verhalten zu unterbinden, können parametrisierte Datentypen eingesetzt werden, die aufgrund ihrer strengen Typisierung durch den *Compiler* verifiziert werden können, zugleich Laufzeitfehler ausschließen und die Ausführungsgeschwindigkeit der Anwendung erhöhen. Die Verwendung dieser generischen Datentypen wird im unteren Block in Auflistung 2.1 ab Zeile 5 dargestellt. Die Typisierung der zweiten Liste erzwingt die ausschließliche Aufnahme von Daten des Typs *Integer* und führt automatisch zu Fehlern im Zuge des Übersetzungsprozesses, sobald ein anderer Datentyp übergeben wird. Ein solcher Fehler, der eine Terminierung der Übersetzung nach sich zieht, wird beispielsweise durch das Hinzufügen des *String*-Datums in Zeile 7 ausgelöst.

Neben den Vorteilen der starken Typisierung bei der Vermeidung von Laufzeitfehlern erhöhen generische Datenstrukturen die Ausführungsgeschwindigkeit. Dieses Phänomen ist in der Generierung einer nativen *Integer*-Liste begründet, die im Zuge des ersten Übersetzungsschrittes aus der Hochsprache in die *Microsoft Intermediate Language (MSIL)* [39] vollzogen wird. Dieses Verfahren spart Rechenzeit, indem die teuren Typumwandlungen vermieden werden, die entstehen, wenn ein Objekt aus einer Liste entnommen und in einer Variablen des Typs *Integer* abgelegt werden muss.

Definition generischer Datentypen

Das Konzept der generischen Datenstrukturen ermöglicht die Entwicklung von benutzerdefinierten Typen, die über einen oder mehrere generische Typparameter verfügen. Die generischen Erweiterungen sind auf Klassen, Schnittstellen, *Structs*, Methoden und Deleganten anwendbar. Die Syntax der generischen Datentypen ist für Klassen, *Structs* und Schnittstellen in Abbildung 2.1 dargestellt. Die Varianten für die Methoden- und Delegantendeklaration sind in der *C#*-Sprachspezifikation [70] nachzulesen.

attributes$_{opt}$
[*class/struct/interface*]-*modifiers*$_{opt}$ partial$_{opt}$ [class/struct/interface] *identifier*
type-parameter-list$_{opt}$ [*class/struct/interface*]-*base*$_{opt}$
type-parameter-contraints-clauses$_{opt}$ [*class/struct/interface*]-*body* ;$_{opt}$

Abbildung 2.1: Deklaration generischen Klassen, Structs und Interfaces [70]

Jedes generische Grundgerüst kann eine unbeschränkte Anzahl Typparameter definieren, wobei jeder Parameter unterschiedlichen Einschränkungen unterliegen kann. Die vollständige Liste der Parametereinschränkungen ist in Tabelle 2.1 dargestellt. Die Einschränkungen ermöglichen beispielsweise festzulegen, dass der bei der Deklaration angegebene Typ eine spezielle Schnittstelle implementiert, von einer Basisklasse erbt oder über einen Standardkonstruktur verfügt.

Einschränkung	**Beschreibung**
where T: struct	Typ T muss ein Werttyp sein.
where T: class	Typ T muss ein Referenztyp sein.
where T: new()	Typ T muss über einen öffentlichen, parameterlosen Konstruktor verfügen
where T : <base class name>	Typ T muss von dem angegebenen Typen erben
where T : <interface name>	Typ T muss das angegebene Interface implementieren

Tabelle 2.1: Einschränkungen von Typparametern [77]

2.2 Operatoren

Die Programmiersprache *C#* unterstützt eine Vielzahl an Operatoren [38], die bezüglich ihrer Operandenanzahl klassifiziert werden. Es werden dabei unäre, binäre sowie ternäre Operatoren unterschieden.

Die Evaluation eines Ausdrucks, bestehend aus Operanden und Operatoren, erfolgt anhand der in der Sprachgrammatik definierten Präzedenz und Assoziativität. Die Operanden eines Ausdrucks werden von links nach rechts evaluiert, wobei die rechtsassoziativen Zuweisungs- und Bedingungsoperatoren eine Ausnahme bilden. Bei Ausdrücken mit mehreren unterschiedlichen Operatoren bindet der Operator mit der höchsten Präzedenz am stärksten und ist somit vorrangig. Bei gleicher Präzedenz entscheidet jedoch die Assoziativität. Die vollständige Liste aller verfügbaren, nach absteigender Präzedenz sortierten Operatoren ist in Tabelle 2.2 dargestellt.

Nr.	Kategorie	Operator
1	Primär	x.y f(x) a[x] x++ x-- new typeof default checked unchecked delegate
2	Unär	+ - ! ~ ++x --x (T)x
3	Multiplikativ	* / %
4	Additiv	+ -
5	Schiebe	<< >>
6	Relational und Typtest	< > <= >= is as
7	Gleichheit	== !=
8	Logisches UND	&
9	Logisches XOR	^
10	Logisches OR	|
11	Bedingtes UND	&&
12	Bedingtes OR	||
13	Null Prüfend	??
14	Bedingung	?:
15	Zuweisung	= *= /= += -= <<= >>= &= ^= |=
16	Lambda Ausdruck	=>

Tabelle 2.2: Operatoren der Sprache C# [38]

Das *Microsoft .NET Framework* implementiert die in Tabelle 2.2 dargestellten Operatoren für alle Basistypen der Klassenbibliothek und ermöglicht dabei die Verwendung der Operatoren in jeglichen Ausdrücken. Die Verwendung von Operatoren ist im Allgemeinen gegenüber der Nutzung von Methoden mit gleicher Semantik deutlich kompakter und leichter verständlich. Das folgende Beispiel in Auflistung 2.2 veranschaulicht dieses Verhalten durch den Vergleich der Darstellung des sehr kompakten Ausdrucks in Zeile 1 und des komplex anmutenden Methodenkonstruktes in Zeile 2.

```
1    double i1 = (10 + 7) * 3 / 100;
2    double i2 = Div(Mult(Add(10, 7), 3), 100);
```

Auflistung 2.2: Operatoren- vs. Funktionsaufrufdarstellung in C#

Überladen von Operatoren

Die Verwendung von Operatoren ist nicht ausschließlich den Basistypen vorbehalten. Benutzerdefinierte Typen können daher ebenfalls von Operatoren profitieren, indem sie diese überladen. Das *Microsoft .NET Framework* schränkt jedoch die Liste der überladbaren Operatoren ein. Die Auflistung der für die benutzerdefinierte Überladung zulässigen unären und binären Operatoren ist in der folgenden Tabelle 2.3 hinterlegt.

Nr.	Kategorie	Operator
1	Unär	+ - ! ~ ++ -- true false
2	Binäre	+ - * / % &= | & ^= << >> == != < > <= >=

Tabelle 2.3: Überladbare Operatoren

Die Implementierung von benutzerdefinierten Operatoren ist an verschiedene Bedingungen geknüpft, die im Detail in [70] nachzuschlagen sind. Es gilt jedoch sowohl für unäre als auch für binäre Operatoren, dass mindestens ein Operand vom Datentyp des den Operator umschließenden Typs sein muss. Zudem erfordert das *Microsoft .NET Framework* im Zusammenspiel mit der Sprache *C#* häufig das Überladen von Operatorpaaren. Dies ist beispielsweise bei der benutzerdefinierten Implementierung des *Gleich*-Operators der Fall, da dieser nur gemeinsam mit dem *Ungleich*-Operator überladen werden kann.

Die Auflistung 2.3 zeigt die Implementierung des gegebenen Beispiels und stellt sowohl den *Gleich*- als auch den *Ungleich*-Operator dar. Die beiden Operatoren vergleichen die über die Parameterlisten in den Zeilen 1 und 6 übergebenen Parameter auf Referenzgleichheit beziehungsweise -ungleichheit und geben das Ergebnis des Vergleichs als booleanschen Wert zurück.

Die als statische Konstrukte implementierten Operatoren zeichnen sich durch die Angabe des Schlüsselwortes *operator* sowie die Angabe des Operatortyps aus. Dieser wird durch das Operatorzeichen repräsentiert. Im Fall des vorliegenden Beispiels wird der Operator durch die Zeichenkette == beziehungsweise ! = angegeben.

```
1    public static bool operator ==(A a1, A a2)
2    {
3       return Object.ReferenceEquals(a1, a2);
4    }
5
6    public static bool operator !=(A a1, A a2)
7    {
8       return !Object.ReferenceEquals(a1, a2);
9    }
```

Auflistung 2.3: Gleich- und Ungleich-Operator

Das Konzept des Überladens der Operatoren ermöglicht es jedem Operator, eine eigens definierte Semantik zu übertragen und den Operator damit optimal auf den gewünschten Anwendungszweck abzustimmen.

2.3 Deleganten

Funktionszeiger sind ein weit verbreitetes Konzept, das in vielen Programmiersprachen Einzug erhalten hat. Sie werden beispielsweise in der Programmiersprache *C++* für die Implementierung von Rückrufmethoden eingesetzt. Die Designer und Entwickler der Sprache *C#* haben das Konzept der Funktionszeiger adaptiert und die als Deleganten [8] bezeichneten Funktionszeiger in die Sprache eingeführt. Diese bilden beispielsweise die Grundlage für die Implementierung von Rückrufmethoden sowie die Umsetzung der Ereignisbehandlung.

```
1    public delegate int CustomDelegate(int i1, int i2);
```

Auflistung 2.4: Deklaration eines typsicheren Funktionszeigers

Die Signatur eines Deleganten wird in Auflistung 2.4 exemplarisch dargestellt. Das Schlüsselwort *delegate* gibt an, dass es sich bei der folgenden Signatur um einen Funktionszeiger handelt. Der vorgestellte Zeiger namens *CustomDelegate* definiert in diesem Beispiel eine Parameterliste mit zwei Parametern des Typs *Integer* sowie einen Rückgabewert desselben Typs.

Die in der Sprache *C#* implementierten Funktionszeiger unterscheiden sich signifikant von der aus *C++* bekannten Variante. Während es sich in der *C++*-Welt ausschließlich um einen Zeiger auf eine Speicheradresse handelt, stellt *C#* ein auf

dem *Heap* verwaltetes Objekt bereit, das sicherstellt, dass der Zeiger auf eine Methode mit einer geeigneten Signatur verweist. Das verwaltete Objekt basiert auf einer *Compiler*-generierten Klasse, die vom Datentyp *System.MulticastDelegate* erbt. Die weitere Betrachtung der Vererbungshierarchie verdeutlicht die Abstammung der *MulticastDelegate*-Klasse vom Datentyp *System.Delegate*.

```
1   // begin of class CustomDelegate
2   .class auto ansi sealed nested public CustomDelegate
3         extends [mscorlib]System.MulticastDelegate
4   {
5      // begin of method CustomDelegate::.ctor
6      .method public hidebysig specialname rtspecialname
7         instance void .ctor(object 'object',
8                             native int 'method') runtime managed
9      {
10     }
11
12     // begin of method CustomDelegate::BeginInvoke
13     .method public hidebysig newslot virtual
14        instance class [mscorlib]System.IAsyncResult
15        BeginInvoke(int32 i1,
16                    int32 i2,
17                    class [mscorlib]System.AsyncCallback callback,
18                    object 'object') runtime managed
19     {
20     }
21
22     // begin of method CustomDelegate::EndInvoke
23     .method public hidebysig newslot virtual
24        instance int32  EndInvoke(class [mscorlib]System.IAsyncResult result)
              runtime managed
25     {
26     }
27
28     // begin of method CustomDelegate::Invoke
29     .method public hidebysig newslot virtual
30        instance int32  Invoke(int32 i1,
31                               int32 i2) runtime managed
32     {
33     }
34  }
```

Auflistung 2.5: Intermediate Language Code des in 2.4 definierten Deleganten

Die Besonderheit der Vererbungshierarchie sowie der Generierung von *Delegate*-Klassen durch den *C#-Compiler* verdeutlicht sich in der Attributierung der Basisklassen. Der *Compiler* verhindert durch die Auswertung der an die beiden Basisklassen angefügten Attribute, dass eine andere Entität als der *Compiler* selbst von diesen Klassen erben darf. Es ist somit ausgeschlossen, manuell eine von *MulticastDelegate* oder *Delegate* erbende Klasse zu implementieren.

Die Auflistung 2.5 zeigt den aus dem Grundgerüst der Deleganten aus Auflistung 2.4 generierten *MSIL*-Quelltext, der in der zugehörigen *Assembly* [91] abgelegt wurde. Die so erzeugte *CustomDelegate*-Klasse verfügt über einen Konstruktor sowie die drei Methoden *Invoke*, *BeginInvoke* und *EndInvoke*, wobei die erste der drei Methoden für die synchrone Verarbeitung und die beiden letztgenannten Methoden für die Verarbeitung mittels des asynchronen, nicht-blockierenden Programmiermodells konzipiert wurden.

Es ist abschließend anzumerken, dass der *C#-Compiler* ausschließlich *Multicast*-Deleganten erzeugt und somit mehr als einen Zeiger auf eine Funktion aufnehmen kann. Dieses Konzept wird beispielsweise ausgiebig bei der Implementierung von *Events* angewendet, da auf jedem *Event* eine Vielzahl von *Handlern* registriert werden soll.

2.4 Zeitgeber

Das *Microsoft .NET Framework* offeriert die drei unterschiedliche Zeitgeberkonzepte *System.Threading.Timer*, *System.Timers.Timer* sowie *System.Windows.Forms.Timer*, die im folgenden klassifiziert und beschrieben werden.

Das *System.Threading.Timer* Konzept basiert auf den Zeitgebern der zugrundeliegenden Hardware und bietet damit die exakteste und zugleich performanteste Implementierung der drei Varianten. Der in Auflistung 2.6 beschriebene Zeitgeber erwartet als Parameter einen *TimerCallback*-Delegaten, der nach Ablauf eines gegebenen Zeitintervalls von einem *Thread-Pool-Thread* aufgerufen wird. Die Rückrufmethode des Zeitgebers wird somit in einem anderen *Thread* ausgeführt als der Zeitgeber selbst. Dieses Verhalten zieht beispielsweise beim Zugriff der Rückrufmethode auf Steuerelemente der Benutzeroberfläche etwaige Sychronisierungsmechanismen, in diesem Fall zwischen dem aufrufenden und dem *GUI-Thread*, nach sich.

Der *System.Timers.Timer* wurde entwickelt, um die *Thread*-Synchronisierung für den Anwendungsentwickler zu vereinfachen. Dieser Zeitgeber wandelt den auf Rückrufmethoden basierenden Ansatz des *System.Threading.Timer* in ein *Event*-gesteuertes Verfahren. Der *System.Timers.Timer* vereinfacht zudem die Synchronisierung zwischen dem *Thread*, der den *Timer* instantiiert und dem *Thread*, in dem die *Timer-Events* ausgelöst werden. Die Synchronisierung der beiden *Threads* erfolgt über die Angabe eines Synchronisierungsobjektes, das über das *Property Syn-*

```
1    public void CreateTimer()
2    {
3       Timer timer = new Timer(TimerCallback, null, dueTime, period);
4    }
5
6    private void TimerCallback(object obj)
7    {
8       // will be executed in an arbitrary Thread-Pool Thread
9    }
```

Auflistung 2.6: System.Threading.Timer

chronizingObject gesetzt werden kann. Insofern das zugewiesene Objekt von der Klasse *Control* erbt, synchronisiert sich der *Timer-Thread* vor der Ausführung des *Timer-Events* mit dem *GUI-Thread* (siehe Kapitel 2.5) und führt das *Event* im Kontext dieses *Threads* aus. Andernfalls wird das *Timer-Event* analog zum *System.Threading.Timer* im Kontext des *Timer-Threads* ausgeführt.

Die Arbeitsweise des *System.Timers.Timer* wird in Auflistung 2.7 dargestellt, wobei die Verwendung des *SynchronizingObject-Properties* in Zeile 6 vorgestellt wird. Das *Property* erwartet ein Synchronisierungsobjekt des Typs *System.ComponentModel.ISynchronizeInvoke*, das beispielsweise bei jedem *Windows-Forms*-Steuerelement enthalten ist und somit eine automatische Synchronisierung zwischen Zeitgeber und graphischer Benutzeroberfläche ermöglicht.

```
1    public void CreateTimer()
2    {
3       Timer timer = new Timer(interval);
4       timer.Elapsed += new ElapsedEventHandler(timer_Elapsed);
5       // SynchronizingObject must implement "System.ComponentModel.
            ISynchronizeInvoke"
6       timer.SynchronizingObject = this.synchronizingObject;
7       timer.Start();
8    }
9
10   private void timer_Elapsed(object sender, ElapsedEventArgs e)
11   {
12      // will be executed in the SynchronizingObject-Thread
13   }
```

Auflistung 2.7: System.Timers.Timer

Das dritte und letzte Zeitgeberkonzept wird vom *System.Windows.Forms.Timer* bereitgestellt. Diese Variante unterscheidet sich signifikant von den beiden vorher beschriebenen Zeitgebern, da der *System.Windows.Forms.Timer*

Eigenschaft	Threading	Timers	Forms
Thread des Timer-Ereignisses	Worker	GUI & Worker	GUI
Synchronisierungsunterstützung	Nein	Ja	Ja
Windows Form erforderlich	Nein	Nein	Ja
Vererbbar	Nein	Ja	Ja
Thread-Sichere Timer-Instanz	Nein	Ja	Nein
Genauigkeit der Ereignisse	Genau	Genau	Ungenau
Begrenzte Systemressource	Ja	Ja	Nein

Tabelle 2.4: Vergleich der drei Zeitgeberkonzepte [69]

nur in Verbindung mit Steuerelementen eingesetzt werden kann, die die Schnittstelle *System.ComponentModel.IContainer* implementieren. Der *System.Windows.Forms.Timer* arbeitet im Vergleich zu den beiden vorherigen Varianten am inexaktesten, besticht aber durch die Übernahme der Synchronisierung durch das Betriebssystem. Dieses wird erreicht, da der Zeitgeber nach Ablauf des definierten Zeitintervalls eine Nachricht in die Windows-Nachrichtenschleife legt, die von den eigenen Steuerelementen analog zu allen weiteren Nachrichten der Warteschlange verarbeitet wird. Da sowohl das Empfangen als auch das Verarbeiten der Nachrichten im *GUI-Thread*, erfolgen ist die Anwendung von Synchronisierungsmethoden nicht erforderlich. Die folgende Auflistung 2.8 verdeutlicht die einfache Verwendung des *System.Windows.Forms.Timer* und den Verzicht auf jegliche Synchronisierungsmechanismen.

```
1   public void CreateTimer()
2   {
3       // Container must implement "System.ComponentModel.IContainer"
4       Timer timer = new Timer(this.container);
5       timer.Tick += new EventHandler(timer_Tick);
6       timer.Interval = this.interval;
7       timer.Start();
8   }
9
10  private void timer_Tick(object sender, EventArgs e)
11  {
12      // will be executed in the Container-Thread
13  }
```

Auflistung 2.8: System.Windows.Forms.Timer

Die Wahl des geeigneten Zeitgebers ist abhängig von der Einsatzdomäne und kann anhand der Tabelle 2.4 getroffen werden. Zusammenfassend lässt sich anmerken,

dass der *System.Threading.Timer* das exakteste Taktungsverhalten zeigt,jedoch den Entwickler bei der *Thread*-Synchronisierung nicht unterstützt und zudem ressourcentechnisch beschränkt ist und somit keine beliebig große Anzahl dieser Zeitgeber simultan verwendet werden kann. Der *System.Timers.Timer*, der denselben Ressourcenbeschränken unterliegt wie die erstgenannte Zeitgebervariante, bietet einen vergleichsweise hohen Synchronisierungskomfort bei ähnlich exaktem Taktungsverhalten. Abgeschlossen wird die Familie der Zeitgeber durch den *System.Windows.Forms.Timer*, der besonders für die Interaktion mit graphischen Steuerelementen prädestiniert ist und nahezu keine *Hardware*-Ressourcen allokiert, dafür jedoch Defizite in der Exaktheit aufweist.

2.5 Thread Synchronisierung mit graphischen Steuerelementen

Graphische Benutzeroberflächen werden mit der Zielsetzung der maximalen Reaktions- und Interaktionsfähigkeit entwickelt. Die Steuerelemente aller verbreiteten Bibliotheken verzichten daher aus Gründen der Performanz auf die Implementierung von *Thread*-Synchronisationsmechanismen und fordern stattdessen, die Ausführung und Manipulation der Steuerelemente auf einen *Thread*, den *Thread* des *Graphical User Interfaces (GUI)*, zu beschränken. Um die Reaktions- und Interaktionsfähigkeit von Anwendungen mit graphischer Benutzeroberfläche zu erhalten, ist es unumgänglich, langlaufenden Operationen in *Worker-Threads* auszulagern und die Resultate im Anschluss an die Berechnung in der Oberfläche darzustellen. Die aus einem *Worker-Thread* resultierenden Ergebnisse können jedoch aufgrund der genannten Einschränkungen nicht direkt in der graphischen Oberfläche dargestellt werden, da dazu eine vorherige Synchronisierung des *Worker-Threads* mit dem *GUI-Thread* erforderlich ist.

Die Synchronisierung eines beliebigen *Worker-Threads* mit dem *Thread* der graphischen Benutzeroberfläche wird im *Microsoft .NET Framework* sowie in der gesamten *Windows API* über den Umweg der *Windows*-Nachrichtenschleife realisiert. Die Nachrichtenschleife ist der zentrale Kommunikationsmechanismus zwischen dem Betriebssystem und den graphischen Anwendungen. Das Betriebssystem transferiert über diesen Kommunikationspfad beispielsweise Informationen über die aktuelle Mausposition oder das Drücken einer Taste auf der Tastatur an die in der Nachrichtenschleife registrierten Applikationsfenster. Die Nachrichten der Schleife werden von den Anwendungen an einem zentralen Punkt entgegengenommen und

weiterverarbeitet. Im Falle der Anwendungen, die auf Basis des *Microsoft .NET Frameworks* entwickelt werden, laufen die Nachrichten in der Methode *WndProc* in jeder von *System.Windows.Forms.Form* erbenden Klasse zusammen und werden im Kontext des *Threads* der graphischen Benutzeroberfläche verarbeitet.

Der Synchronisierungsmechanismus erfordert, dass der *Worker-Thread* eine Nachricht in die *Windows*-Nachrichtenschleife einstellt, die innerhalb der *WndProc*-Methode im Kontext des *GUI-Threads* verarbeitet wird. Die eingestellte Nachricht beinhaltet einen Funktionszeiger auf die aufzurufende Methode sowie die zu übergebenden Parameter. Das *Microsoft .NET Framework* kapselt das skizzierte Vorgehen in der *Invoke*-Methode, die in jedes Steuerelement vererbt wird.

```
1   public class InvokeSample : Form
2   {
3      private int tickCounter = 0;
4
5      public InvokeSample()
6      {
7         Threading.Timer timer = new Threading.Timer(TimerCallback, null, 1000,
              1000);
8      }
9
10     private void TimerCallback(object obj)
11     {
12        if (this.InvokeRequired)
13           this.Invoke(new Threading.TimerCallback(TimerCallback), obj);
14        else
15           this.Text = (++tickCounter).ToString();
16     }
17  }
```

Auflistung 2.9: Invoke am Beispiel der Klasse System.Windows.Forms.Form

Das vorgestellte Verfahren wird am Beispiel eines *System.Threading.Timers* in Auflistung 2.9 dargestellt. Der verwendete *Timer* ruft in einem Zeitintervall von 1.000 ms die Methode *TimerCallback* im Kontext eines beliebigen *Thread-Pool-Threads* auf. Ein direkter Zugriff auf die graphische Oberfläche ist somit nicht zulässig. Die Prüfung, ob eine Methode im richtigen Kontext und somit im *GUI-Thread* aufgerufen wird, erfolgt über das *Property InvokeRequired*, dessen Verwendung in Zeile 12 des Beispiels aufgezeigt wird. Insofern eine Synchronisierung erforderlich ist, wird die Methode *Invoke* der *InvokeSample*-Klasse aufgerufen. Dies erwartet als Parameter einen Funktionszeiger auf die Methode, die im Kontext des *GUI-Threads* aufgerufen werden soll, sowie die Parameter, die dieser Methode übergeben werden sollen. Nachdem die *Invoke*-Anweisung in Zeile 13 ausgeführt wurde, wird eine Nachricht in die

Warteschlange eingestellt und im *GUI-Thread* verarbeitet. Dieser Prozess ruft die *TimerCallback*-Methode ein zweites mal auf, das *InvokeRequired-Property* evaluiert *falsch*, und die Ausführung wird in Zeile 15 mit dem Zugriff auf das *Text-Property* der Oberfläche fortgesetzt.

2.6 Iteratoren

Iteratoren sind ein weit verbreitetes Konzept in der Welt der objektorientierten Programmiersprachen. Sie erlauben es, sequentiell über die Elemente einer Menge, beispielsweise einer Liste, zu traversieren, ohne dabei die zugrundeliegenden Datenstrukturen zu betrachten. Das Architekturmodell ist mit zwei Elementarbausteinen dahingehend konzipiert, dass Iteratoren in verschachtelten Schleifen oder parallelen *Threads* durchlaufen werden können. Der erste Baustein, der Generator, ist für die Produktion der Elementsequenzen zuständig, während der zweite Baustein einen *Container* bereitstellt, der für jeden Konsumenten einen eigenen Generator erzeugt. Die Vorteile dieser zweigeteilten Architektur werden anhand der verschachtelten Schleife in Auflistung 2.10 dargestellt. Das *Container*-Objekt erzeugt sowohl für die äußere als auch für die innere *foreach*-Schleife in den Zeilen 4 beziehungsweise 6 einen eigenen, unabhängigen Generator und ermöglicht somit die vollständig disjunkte Traversion über die identische zugrundeliegende Menge.

```
1   public void NestedLoop()
2   {
3       string[] elements = new string[] { "e1", "e2", "e3", "e4" };
4       foreach (string elementLoop1 in elements)
5       {
6           foreach (string elementLoop2 in elements)
7           {
8               Console.WriteLine("{0} - {1}", elementLoop1, elementLoop2);
9           }
10      }
11  }
```

Auflistung 2.10: Verschachtelte *foreach*-Schleife

Microsoft adaptierte dieses aus Python [122] bekannte Konzept der Iteratoren und führte es unter der Bezeichnung *Enumerator* in das *.NET Framework* und die Sprache *C#* ein. Die Entwicklung eines *Enumerators* setzt die Implementierung der Schnittstelle *System.Collections.IEnumerator* für den Generator und *System.Collections.IEnumerable* für das *Container*-Objekt voraus. Dieses sehr aufwändige und oftmals redundante Verfahren wurde mit Vorstellung der zweiten Version

des *Microsoft .NET Frameworks* signifikant vereinfacht. Die Einführung von Iteratorblöcken und der *yield*-Anweisung ermöglichen es dem *C# Pre-Compiler*, automatisiert Klassen zu erzeugen, die die Schnittstelle *IEnumerator* und optional *IEnumerable* implementieren und somit den Entwicklungsaufwand auf ein Minimum reduzieren. Die Implementierung eines Iteratorblockes folgt dabei der in Abbildung 2.2 dargestellten Deklarationsvorschrift. Die Abbildung offenbart zudem, dass mit der Einführung der generischen Datentypen in Version 2.0 des *Microsoft .NET Frameworks* das Objektmodell der *Enumeratoren* um die beiden generischen Pendants *IEnumerator<T>* und *IEnumerable<T>* der vorgestellten Schnittstellen erweitert wurde.

attributes$_{opt}$ *method-modifiers*$_{opt}$
return-type [*Enumerator/IEnumerable/IEnumerator<T>/IEnumerable<T>*]
identifier type-parameter-list$_{opt}$ (*formal-parameter-list*$_{\text{opt}}$)
type-parameter-contraints-clauses$_{opt}$;

Abbildung 2.2: Iteratordeklaration

Eine Methode wird im Kontext des *Microsoft .NET Frameworks* als Iteratorblock bezeichnet, wenn sie der vorgestellten Deklarationssyntax aus Abbildung 2.2 entspricht. Ein solcher Block ist in seinem Rückgabetyp festgelegt. Dieser muss *IEnumerator*, *IEnumerable* oder den generischen Varianten *IEnumerator<T>* beziehungsweise *IEnumerable<T>* entsprechen. Die Typen der vom Generator produzierten Werte entsprechen im Fall der generischen Schnittstellen dem Typparameter T und im Fall der nicht-generischen Schnittstellen dem Typ *Object*.

Die automatisierte Generierung der *Enumeratorklassen* durch den *C#-Compiler* bedingt jedoch einige Einschränkungen für die Iteratorblöcke. Daher unterscheiden sich diese trotz syntaktischer Äquivalenz signifikant von normalen Methoden. Ein Iteratorblock wird durch den *Compiler* in eine Klasse umgewandelt. Der Block wird aus der umschließenden Klasse entfernt und durch ein *Property* vom Typ der erzeugten Klasse ersetzt. Der Iteratorblock ist damit nicht mehr Bestandteil der definierenden Klasse und unterliegt somit einigen Beschränkungen. Beispielsweise ist die Verwendung von Werttypen per Referenz in der Parameterliste untersagt. Somit ist die Deklaration von *ref* beziehungsweise *out* Parametern innerhalb der Parameterliste eines Iteratorblocks nicht zulässig und führt genauso wie die Verwendung der *return*-Anweisung zu einem Compiler-Fehler. Letztere ist durch die Verwendung der *yield return*-Anweisung zur Produktion eines neuen Elementes oder durch die Anweisung *yield break* zum Ausstieg aus einem Iterator zu ersetzen. Abschließend ist

anzumerken, dass sich kein als *unsafe* gekennzeichneter Quellcode innerhalb eines Iterators befinden darf, der Iterator selbst jedoch Bestandteil eines solchen Blockes sein kann.

Zugriff auf einen Enumerator

Der Zugriff auf einen *Enumerator* beziehungsweise dessen *Container*-Objekt kann sowohl implizit als auch explizit erfolgen. Beide Zugriffsvarianten werden im folgenden Beispiel in Auflistung 2.11 anhand eines *Enumerators*, bestehend aus einem Generator mit zugehörigem *Container*-Objekt, vorgestellt, der eine Sequenz von *String*-Datentypen erzeugt.

Die implizite Verwendung des *Enumerators* wird in den Zeilen 10 und 11 der Auflistung dargestellt. Bei der Initialisierung der *foreach*-Schleife wird mittels des *IEnumerable<string>*-Containers ein neuer Generator vom Typ *IEnumerator<string>* erzeugt, über den die Schleife traversiert. Der *Enumerator* wird von der *foreach*-Schleife insgesamt viermal durchlaufen. Bei den ersten drei Iterationen wird jeweils ein Datum erzeugt und über die Ausgabefunktion auf die Kommandozeile geschrieben. Nachdem die Werte *Hello*, *World* und *42* ausgegeben wurden, wird der *Enumerator* ein viertes mal durchlaufen. Dieser terminiert jedoch sofort, da kein weiteres Datum produziert werden kann.

```
1    public IEnumerable<string> Iterator()
2    {
3       yield return "Hello";
4       yield return "World";
5       yield return "42";
6    }
7
8    public void Print()
9    {
10      //Implicit implementation
11      foreach (string str in Iterator())
12         Console.WriteLine(str);
13
14      //Explicit implementation
15      IEnumerator<string> iter = Iterator().GetEnumerator();
16      while (iter.MoveNext())
17         Console.WriteLine(iter.Current);
18   }
```

Auflistung 2.11: Implizite und explizite Verwendung eines Iterators

Analog zur impliziten Variante wird die explizite Verwendung ab Zeile 15 dargestellt. Das Verfahren beginnt mit der Erzeugung eines Generators durch den Aufruf

der Methode *GetEnumerator*. Dieser verfügt über die Methode *MoveNext*, die den booleanschen Wert *wahr* zurückgibt, solange der Generator neue Werte produzieren kann. Die Rückgabewert der Methode kann somit als Abbruchbedingung für die *while*-Schleife in Zeile 16 genutzt werden. Im vorliegenden Beispiel wird die *MoveNext*-Methode viermal aufgerufen. Sie liefert bei den ersten drei Durchläufen den Wahrheitswert *wahr* und im letzten Durchlauf den Wert *falsch*. Die Ausgabe des produzierten Datums erfolgt in jeder Schleifeniteration über das *Property Current*, indem das aktuell produzierte Datum abgelegt wird.

Trotz der unterschiedlichen Darstellung des impliziten und expliziten Zugriffs auf einen *Enumerator* sind beide Varianten als äquivalent zu betrachten. Dieses resultiert aus dem Umstand, dass der *C#-Compiler* jede *foreach*-Schleife in eine *while*-Schleife wandelt und der in der *Assembly* abgelegte *MSIL*-Code der beiden Varianten somit nahezu identisch ist.

Konzept der Enumeratorgenerierung

Die automatisierte Generierung der *Enumeratoren* wird durch die Umwandlung von Iteratorblöcken in *Enumeratorklassen* durch den *C# Pre-Compiler* erreicht. Das Verfahren der manuellen Implementierung folgt dem automatisierten mit Ausnahme der Quellcodegenerierung und wird daher im folgenden nicht behandelt. Eine ausgiebige Darstellung des manuellen Verfahrens kann unter [125] nachgelesen werden.

Der *C# Pre-Compiler* entscheidet anhand des Rückgabetyps eines jeden Iteratorblockes, welche Klassen generiert werden und welche Schnittstellen diese implementieren. Im Falle der Verwendung der nicht generischen Schnittstellen *IEnumerator* und *IEnumerable* werden diese durch den *Compiler* als generische Schnittstellen mit Typparameter *Object* interpretiert. Ein Iteratorblock mit Rückgabetyp *IEnumerator<T>* erzeugt eine Klasse, die die Schnittstellen *IEnumerator<T>*, *IEnumerator* und *IDisposable* implementiert. Ein Iteratorblock mit Rückgabetyp *IEnumerable<T>*, der neben dem Generator auch noch das optionale *Container*-Objekt erzeugt, implementiert zu den Schnittstellen *IEnumerator<T>*, *IEnumerator* und *IDisposable* für den Generator auch noch die Schnittstellen *IEnumerable<T>* und *IEnumerable*.

IEnumerator<T>

Die Signatur der Schnittstelle *IEnumerator<T>* und ihrer Vererbungshierarchie ist

im Klassendiagramm in Abbildung 2.3 dargestellt. Die beiden Varianten der *IEnumerator* Schnittstellen unterscheiden sich zum einen in der Typisierung des *Current-Properties* und zum anderen durch die Implementierung von *IDisposable* im Falle der generischen Schnittstelle. Die ausschließliche Implementierung von *IDisposable* in der generischen Schnittstelle ist historisch bedingt. Auf eine nachträgliche Änderung der Schnittstellensignatur von *IEnumerator* wurde aus Kompatibilitätsgründen verzichtet.

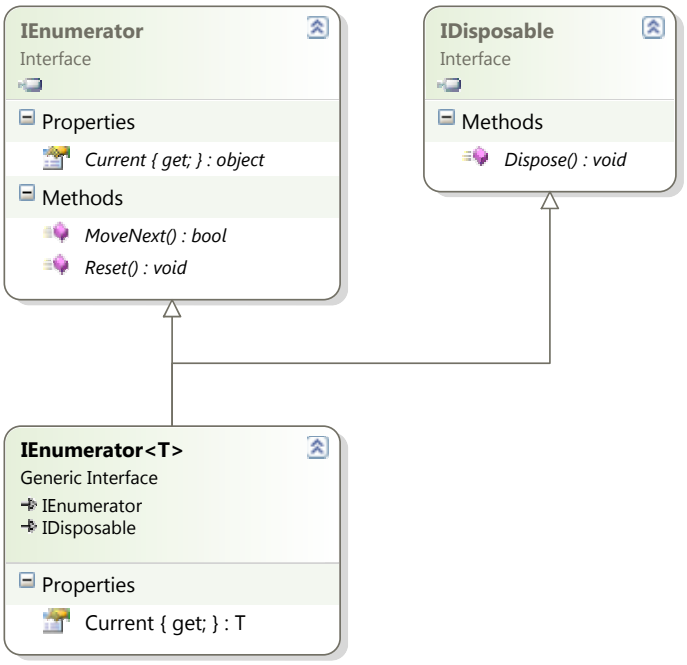

Abbildung 2.3: Diagramm der IEnumerator-Schnittstelle

Bei der Generierung der *Enumeratorklassen* werden die oben genannten Schnittstellen und damit die Methoden *MoveNext*, *Reset* sowie *Dispose* und das *Property Current* implementiert. Bei der automatischen Generierung werden die Methoden *Dispose* und *Reset* jedoch nicht betrachtet. Im Fall der *Dispose*-Methode wird ausschließlich die Methodensignatur mit leerem Methodenrumpf erzeugt. Die *Reset*-Methode wird hingegen mit dem Aufruf *throw new System.NotSupportedException()* ausgestattet, der zur Ausführung einer Ausnahme führt.

Die eigentliche Funktionalität des *Enumerators* wird in der *MoveNext*-Methode im Zusammenspiel mit dem *Current-Property* und einer Zustandsvariablen abgebildet. Der Quellcodegenerator zerlegt den Iteratorblock in einzelne Abschnitte und weist diesem einen eindeutigen nummerischen Zustandsidentifikator zu. Das Verfahren der Umwandlung wird anhand des in den Zeilen 1 bis 6 dargestellten Iteratorblocks aus Auflistung 2.11 aufgezeigt. Der resultierende Quelltext des Umwandlungspro-

zesses ist ausschnittsweise in Auflistung 2.12 dargestellt. Die *MoveNext*-Methode implementiert dazu eine *switch*-Anweisung, die auf einer internen, klassenweiten Zustandsvariablen operiert und bei jedem Aufruf der Methode in die jeweilige Verzweigung springt. Sobald ein *case*-Zweig erreicht ist, wird die Zustandsvariable auf -1 gesetzt. Dieser Wert signalisiert, dass sich der *Enumerator* im Produktionsprozess befindet. Im Anschluss an die Änderung der Zustandsvariablen wird dem *Current-Property* der aktuelle Wert zugewiesen und die interne Zustandsvariable auf den Wert des Folgezustands gesetzt. Der Wert des booleanschen Rückgabetyps der *MoveNext*-Methode ist solange *wahr*, bis kein weiterer Wert mehr produziert werden kann. Im angegebenen Beispiel ist dies der Fall, sobald die interne Zustandsvariable den Wert 3 annimmt.

```
1       private bool MoveNext()
2       {
3           switch (this.state)
4           {
5               case 0:
6                   this.state = -1;
7                   this.current = "Hello";
8                   this.state = 1;
9                   return true;
10
11              case 1:
12                  this.state = -1;
13                  this.current = "World";
14                  this.state = 2;
15                  return true;
16
17              case 2:
18                  this.state = -1;
19                  this.current = "42";
20                  this.state = 3;
21                  return true;
22
23              case 3:
24                  this.state = -1;
25                  break;
26          }
27          return false;
28      }
29
30      string IEnumerator<string>.Current
31      {
32          get { return this.current; }
33      }
```

Auflistung 2.12: Ausschnitt aus der generierten Enumeratorklasse

Kapitel 3

Gears4Net

Die Entwicklung von Anwendungen auf Mehrprozessorsystemen stellt die Methoden und Paradigmen der Softwareentwicklung vor neue Herausforderungen. Das folgende Kapitel beschreibt nach eingehender Analyse bestehender Modelle das Programmiermodell *Gears4Net* [110], das speziell für die Entwicklung massiv paralleler Applikationen konzipiert wurde. Neben der Analyse bestehender Modelle und der Vorstellung der *Gears4Net*-Konzeption wird die Umsetzung des Modells auf Basis des Microsoft .NET Frameworks beschrieben und dessen Nutzung mit Quellcodebeispielen erläutert. Abgeschlossen wird dieses Kapitel mit der Evaluierung des Systems sowie der Abgrenzung gegenüber verwandten Arbeiten.

3.1 Motivation

Die Paradigmen und Konzepte der Softwareentwicklung sind bis heute auf die Implementierung von Applikationen mit einem geringen Parallelisierungsgrad ausgelegt und setzen daher auf die Parallelisierungs- und Synchronisationsmechanismen, die aus den Anfängen der Betriebssystementwicklung stammen. Im Gegensatz zu den Softwareentwicklungsmethoden unterliegt die Entwicklung moderner Hardwarekomponenten rapiden Veränderungen. Die zunehmende Anforderung an die Rechenleistung aktueller Hardware wird nicht mehr ausschließlich eindimensional durch die Erhöhung von Taktraten und Transistoren auf den Chips erreicht, sondern äußert sich zunehmend in einer zweiten Dimension durch die Bereitstellung von *Multi-* und *Many-Core* Architekturen, die eine Vielzahl paralleler Rechenkerne ermöglichen.

Der Trend zur Verwendung aufwändiger Mehrprozessorarchitekturen hat sich ausgehend von spezialisierter und teurer Serverhardware auf den Endkundenmarkt ausgebreitet. Recheneinheiten mit mehreren Kernen finden daher in nahezu jedem aktuel-

len Gerät ihre Anwendung und sind beispielsweise in Notebooks und Spielkonsolen verbaut.

Die optimale Ausnutzung der durch die neue Hardwarearchitektur bereitgestellten parallelen Rechenleistung muss durch neue Softwareentwicklungsparadigmen und Programmiermodelle gewährleistet werden. Die Herausforderungen im Umgang mit der neuartigen Parallelität offenbaren sich eindrucksvoll bei der Entwicklung von Computerspielen auf den aktuellen Spielkonsolen Microsoft XBox 360 [75] und Sony PlayStation 3 [99], deren vollständige Rechenleistung erst nach jahrelanger Erfahrung mit der entsprechenden Plattform abgerufen werden kann. Die Signifikanz dieser Erkenntnis wird für die allgemeine Applikationsentwicklung auf Mehrprozessorsystemen besonders durch die Eigenschaft der Hardwareidentität der Spielkonsolen verdeutlicht. Stellt also bereits die Applikationsentwicklung für eine spezifische Hardwareplattform mit einer definierten Anzahl an Rechenkernen eine überdurchschnittliche Herausforderung an die Entwickler der Applikation, so potenziert sich diese Herausforderung bei der Entwicklung einer Applikation für eine unbekannte und stetig wechselnde Hardwareplattform.

Die Herausforderungen der Applikationsentwicklung auf massiv parallelen Systemen ist auf zwei Problemstellungen zurückzuführen. Erstens unterstützt keines der praxisrelevanten Programmiermodelle die Parallelität als Teil des Softwaredesignprozesses, und zweitens führt die Entwicklung mit diesen Modellen sehr schnell zu Synchronisationsproblemen und im schlimmsten Fall zu ungewünschten Effekten der Nebenläufigkeit während der Applikationsausführung. Als Folge der nicht optimalen Programmiermodelle verdrängt die Problematik der Parallelität die Komplexität der eigentlich zu implementierenden Applikationslogik. [45, 64, 65, 94]

Die Zielsetzung dieses Teils der Arbeit liegt daher in der Konzipierung und Entwicklung eines Programmiermodells, das die Entwicklungskomplexität von massiv parallelen Applikationen deutlich reduziert und zudem möglichst wenige Einschränkungen auf die Auswahl der zugrundeliegenden Entwicklungsplattform beziehungsweise das zugrundeliegende Betriebssystem ausübt.

3.2 Programmiermodellanalyse

Die Auswahl eines Programmiermodells ist von eminenter Signifikanz für die Architektur, Komplexität, Wartbarkeit und Skalierungsfähigkeit eines Softwareprojektes

und bestimmt somit den technischen und betriebswirtschaftlichen Erfolg beziehungsweise Misserfolg eines Projektes. Der Zusammenhang zwischen dem ausgewählten Programmiermodell und der Art der Implementierung des Quelltextes ist offenkundig, da das Modell die Strukturierung, das Verständnis sowie die Wartbarkeit des Quellcodes beeinflusst.

Das Kapitel Programmiermodellanalyse vergleicht daher die vier Modelle der blockierenden, nicht-blockierenden, asynchronen und Aktor-basierten Programmierung, die besonders für die Entwicklung verteilter Systeme relevant sind. Die ersten drei Modelle haben Einfluss in die meisten Programmiersprachen erhalten und sind somit Teil bestehender *Frameworks*. Die Aktor-basierte Programmierung ist hingegen als *Software-Pattern* und somit implementierungsunabhängig zu betrachten. Das Systemmodell, das der Analyse zugrunde liegt, geht von einer massiv parallelen Umgebung aus, wie diese bei der Entwicklung von verteilten Systemen üblich ist. Die Analyse der konträren beziehungsweise teilweise überlappenden Modelle umfasst drei Aspekte, die für jedes der vier Modelle einzeln ausgewertet werden. Der erste Analyseaspekt beinhaltet den effizienten Umgang mit den Hardware- und Software-Ressourcen und den daraus abgeleiteten Skalierungs- und Performanzeigenschaften des Modells. Die Strukturierung des Quelltextes, die in direktem Abhängigkeitsverhältnis zum gewählten Programmiermodell steht, bildet zusammen mit der Erkennung komplexerer Ereignisse den zweiten Analyseaspekt. Der Abschluss der Analyseaspekte wird durch die Synchronisierungsproblematik geprägt und beinhaltet die Fragestellung, wie häufig und transparent die Synchronisierung paralleler *Threads* [16, 17] auf einem Objekt erfolgen muss.

Die folgende Analyse beschreibt die vier Modelle blockierende Programmierung, nicht-blockierende Programmierung, asynchrone Programmierung und Aktor-basierte Programmierung und skizziert die jeweiligen Vor- und Nachteile der vorgestellten Modelle.

3.2.1 Modell I: Blockierende Programmierung

Das Modell der blockierenden Programmierung [14, 19, 113, 119] ist das weit verbreitetste und intuitivste Konzept. Das Betriebssystem teilt dazu jedem *Thread* eines Prozesses nach einem definierten *Scheduling*-Verfahren Rechenzeit zu. Dieses Verfahren wird solange fortgesetzt, bis ein *Thread* einen blockierenden Systemaufruf tätigt. Der aufrufende *Thread* wird solange blockiert, bis der Aufruf zurückkehrt.

Die Blockade eines *Threads* ist gleichbedeutend mit dem Ausschluss desselben aus der Liste des *Schedulers*.

Analyseaspekt 1: Ressourceneffizienz

Misst man das Modell der blockierenden Programmierung am ersten Analyseaspekt, so wird deutlich, dass das Modell weder Rechenzeit- noch Speichereffizient arbeitet. Die ineffiziente CPU-Nutzung des Modells wird nicht durch die Blockade eines oder mehrerer *Threads* hervorgerufen, sondern begründet sich in dem Umstand, dass erstens die Erzeugung eines *Threads* einen Sprung in den *Kernel-Modus* des Betriebssystems erfordert, der erhebliche Kosten verursacht, und zweitens die Betriebssystemarchitektur inklusive der *Scheduling*-Verfahren nicht auf die massive Anzahl an *Threads* ausgelegt ist. Dieses führt beispielsweise dazu, dass die Zeitscheiben, die für jeden *Thread* zur Verfügung stehen, immer kleiner werden und der Verwaltungsaufwand des *Schedulers* steigt.

Die Schwäche im effizienten Umgang mit der Ressource CPU setzt sich beim Verbrauch des Arbeitsspeichers fort. Die Abbildungen 3.1, 3.2 und 3.3 offenbaren zum einen die ineffiziente Speichernutzung und zum anderen die Effekte des Betriebssystems, das nicht zur Nutzung von beliebig vielen *Threads* pro Prozess konzipiert ist. Die drei Diagramme stellen die Erzeugungszeit, den aktuell verwendeten Speicher sowie den angeforderten virtuellen Speicher in Abhängigkeit von der Anzahl der erzeugten *Threads* dar. Die Erzeugungszeit sowie der aktuell verwendete Speicher werden auf der primären Y-Achse, der angeforderte virtuelle Speicher auf der sekundären Y-Achse und die Anzahl der *Threads* auf der X-Achse aufgezeigt.

Die drei Messungen, die den Diagrammen zugrunde liegen, unterscheiden sich in der Rechner- beziehungsweise Betriebssystemarchitektur sowie der Größe des *Stack* pro *Thread*. Der Versuchsaufbau der Messungen ist so konzipiert, dass so viele *Threads* erzeugt werden, wie das zugrundeliegende Betriebssystem zulässt, und die Messung durch das Auslösen einer *OutOfMemoryException* beendet wird. Die erste Messung, deren Resultat in Abbildung 3.1 dargestellt ist, wurde auf einem 32 Bit Windows 7 mit der *Stack*-Größe von *512 KB* aufgenommen. Die Messung zeigt, dass das Betriebssystem maximal *1400 Threads* erzeugt und dabei etwa *1,4 GB* virtuellen Speicher allokiert, während der effektive Speicherbedarf die Marke von *50 MB* nicht überschreitet.

Die zweite der drei Messungen unterscheidet sich von der vorangegangenen ausschließlich in der Veränderung der *Stack*-Größe. Diese wurde von der Windows-Standardgröße von *512 KB* auf den Minimalwert *256 KB* verringert. Das Ergebnis

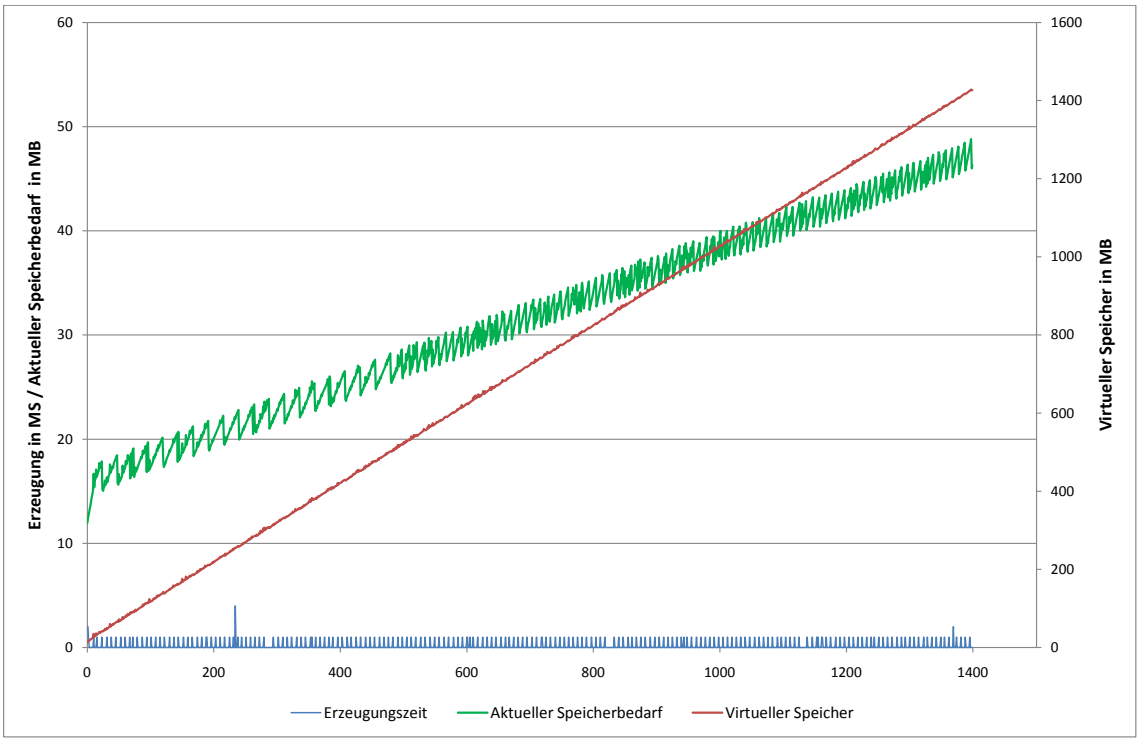

Abbildung 3.1: Thread Erzeugnung - Windows 7 - 32 Bit - Stack: 512 KB (Hardware: Lenovo Thinkpad T60p 2007-8HG, 2 GB RAM)

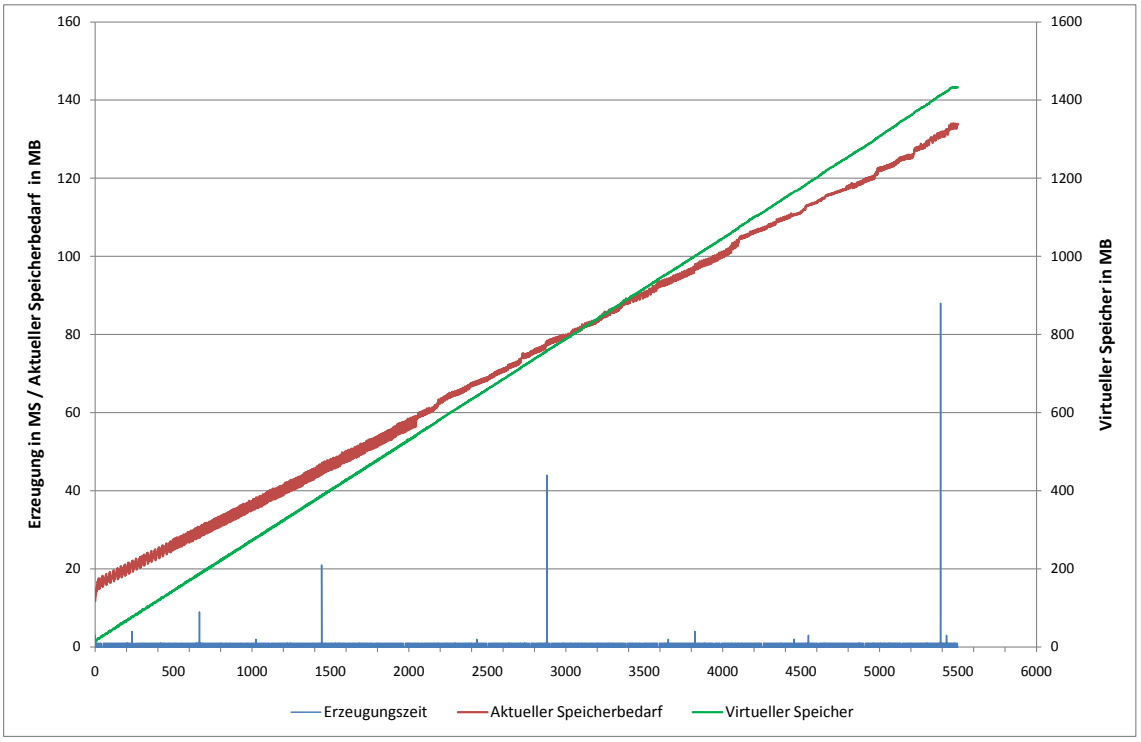

Abbildung 3.2: Thread Erzeugnung - Windows 7 - 32 Bit - Stack: 256 KB (Hardware: Lenovo Thinkpad T60p 2007-8HG, 2 GB RAM)

der Messung überrascht durch die Anzahl der erzeugten *Threads*. Die Halbierung der *Stack*-Größe führt nicht zu einer Verdoppelung der *Thread*-Anzahl, sondern vervierfacht die Anzahl der *Threads* überraschenderweise auf *5500*. Zudem weist die auf der primären Y-Achse aufgetragene Kurve der Erzeugungszeit der *Threads* einige *Peaks* auf, die darauf schließen lassen, dass das Betriebssystem die Erweiterungen der Prozess- beziehungsweise *Thread*-Tabellen vornehmen muss.

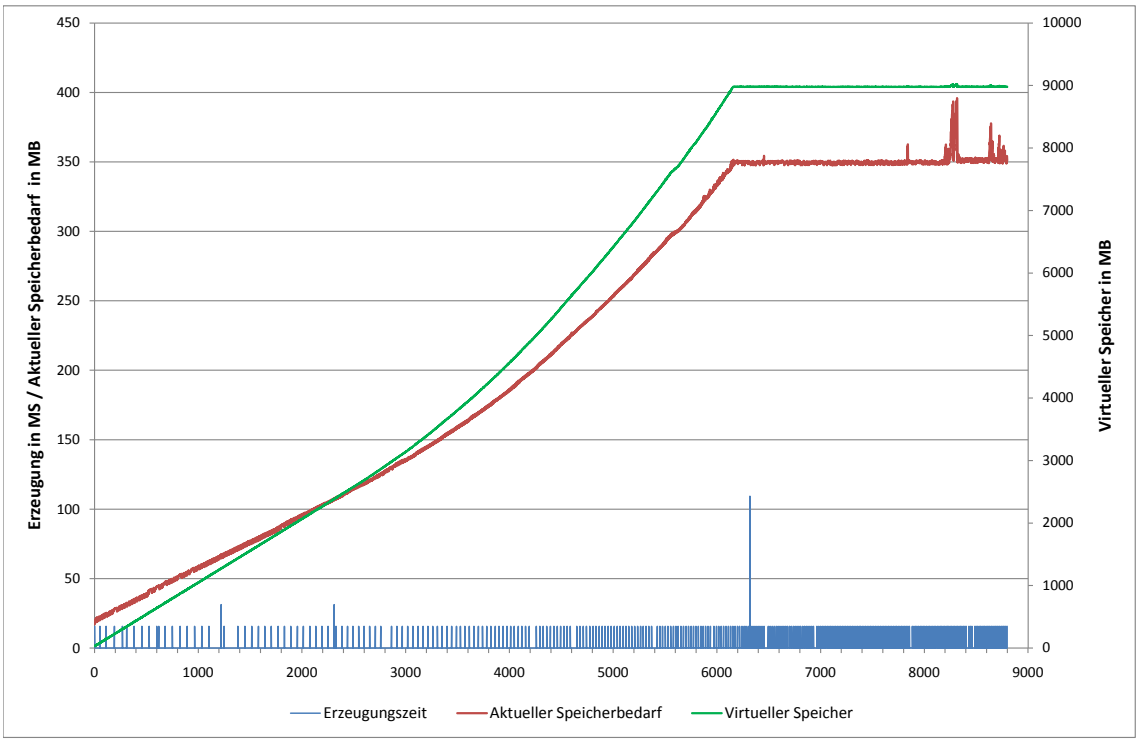

Abbildung 3.3: Thread Erzeugnung - Windows 2008 - 64 Bit - Stack: 512 KB (Hardware: Dell Power Edge 1800 - 6 GB RAM)

Die letzte Messung erfolgte im Gegensatz zu den beiden vorhergehenden Varianten auf einem 64 Bit Rechner, auf dem ein Windows Server 2003 64 Bit ausgeführt wird. Die Konfiguration der *Stack*-Größe ist mit *512 KB* auf den Minimalwert des Betriebssystems festgesetzt. Die beiden Speicherkurven, die in Abbildung 3.3 aufgetragen sind, unterscheiden sich von den beiden vorherigen Messungen durch ihren nichtlinearen Charakter bei der Erzeugung der ersten *6000 Threads* und darüber hinaus durch den konstanten Speicherverbrauch im Bereich von *6000* bis *8800 Threads*. Der konstante Verlauf beider Speicherkurven ist durch das *Thread*-Erzeugungsverfahren begründet. Sobald ein neuer *Thread* erzeugt werden soll, wird eine Anforderung in eine Warteschlange eingestellt, die vom Betriebssystem abgearbeitet wird. Das Betriebssystem nimmt Anforderungen entgegen, solange das System über ausreichend Speicher verfügt. Die Annahme der Anforderungen ist jedoch unabhängig von der

	W7 256 KB	W7 512 KB	W2K8 512 KB
Architektur	x86 - 32 Bit	x86 - 32 Bit	x64 - 64 Bit
Stack Größe	256 KB	512 KB	512 KB
Anzahl Threads	5500	1400	8800
Kumulierte Stack Größe	1375 MB	700 MB	4400 MB
Virtueller Speicher	1433 MB	1426 MB	8978 MB
Verwendeter Speicher	58 MB	46 MB	350 MB
Verwendeter Speicher / Kumulierte Stack Größe	0,097	0,066	0,079
Virtueller Speicher / Kumulierte Stack Größe	1,04	2,03	2,04

Tabelle 3.1: Zusammenfassung der Ergebnisse bei der Erzeugung maximal vieler Threads in unterschiedlichen Konfigurationen

Anzahl der maximal erzeugbaren *Threads*. Im Bezug auf die vorgestellte Messung bedeutet dies, dass das Betriebssystem ab der Grenze von ca. 6000 *Threads* Erzeugungsanfragen zulässt, jedoch keine neuen *Threads* erzeugt. Der Speicherverlauf bleibt daraufhin konstant.

Die Zusammenfassung der Ergebnisse aus den drei Messungen ist in Tabelle 3.1 dargestellt. Die erhobenen Daten zeigen sehr große Divergenzen und lassen Raum für vielschichtige Interpretationen der Ergebnisse. Es ist jedoch festzuhalten, dass die Anzahl der erzeugbaren *Threads* durch den maximal adressierbaren virtuellen Adressraum beschränkt ist. Die Ergebnisse verdeutlichen zudem, dass die Erzeugung von mehreren hunderten oder tausenden *Threads* nicht mit den Konzepten der Betriebssystemarchitekturen in Einklang steht und da schon allein aus diesem Grund vom Modell der blockierenden Programmierung in massiv parallelen und skalierenden Umgebungen Abstand genommen werden muss.

Analyseaspekt 2: Quellcodestrukturierung

Die Struktur des Quelltextes ist von erheblicher Bedeutung für das Verständnis der implementierten Logik und somit ausschlaggebend für die spätere Wartung. Das Modell der blockierenden Programmierung ermöglicht eine sehr simple Darstellung, da durch die blockierenden Systemaufrufe der Quelltext Zeile für Zeile gelesen und verstanden werden kann. Das Modell erfordert in seiner klassischen Darstellung keinerlei Rückrufmethoden oder sonstige Konstrukte, die die Lesbarkeit des Programmtextes erschweren. Die transparente Darstellung stößt allerdings an ihre Grenzen, sobald

eine Zusatzbedingung zu einem blockierenden Systemaufruf hinzugefügt werden soll. Dies ist beispielsweise dann der Fall, wenn ein blockierender Aufruf nach einer definierten Zeit abgebrochen werden soll und der Systemaufruf keine *Timeout*-Angabe zulässt. In einer solchen Konstellation ist die Verwendung einer *Timer*-Instanz unumgänglich, das nach Ablauf des definierten Intervalls den blockierenden Aufruf abbricht. Das beschriebene Szenario würde beispielsweise auf eine der in Kapitel 2.4 beschriebenen *Timer*-Objekte zurückgreifen, die über ein asynchrones Programmiermodell angesprochen werden. Ein weiteres Beispiel, das die Grenzen des Modells der blockierenden Programmierung aufzeigt, ist gegeben, wenn auf mehrere blockierende Operationen gleichzeitig gewartet werden soll. Dieses erfordert die Abspaltung eines *Threads* für jeden parallel auszuführenden blockierenden Systemaufruf sowie die spätere Synchronisierung der unterschiedlichen Ausführungsfäden.

Analyseaspekt 3: Synchronisierung

Der Vorteil der blockierenden Programmierung wird vor allem durch die sequentielle Abarbeitung des Quellcodes deutlich. Sofern nur ein *Thread* zur Ausführung des Programmcodes verwendet wird, erübrigt sich die Fragestellung der erforderlichen Synchronisierung. Folgt der Programmcode jedoch einem der beiden im vorherigen Abschnitt geschilderten Beispiele und sind somit zur gleichen Zeit mehrere *Threads* aktiv, so muss auch bei der blockierenden Programmierung auf die bekannten Synchronisierungskonzepte zurückgegriffen werden.

3.2.2 Modell II: Nicht-blockierende Programmierung

Das Modell der nicht-blockierenden Programmierung [14, 113, 119] ist vorwiegend auf systemnahen Schichten, beispielsweise bei der direkten Micro-Prozessor Programmierung, anzutreffen. Die dem Modell zugrundeliegende Idee besagt, dass eine direkt zurückkehrende Methode solange in einer Schleife aufgerufen wird, bis das Ergebnis vorliegt. Das auch als *Polling* bezeichnete Verfahren lässt sich am Beispiel der Netzwerk-*Socket* Programmierung verdeutlichen. Auf einem *Socket* wird mittels einer *do-while*-Schleife das Property *Available* abgerufen, bis der Rückgabewert der Funktion größer als *0* ist. Sobald die Bedingung erfüllt ist, liegt am *Socket* ein Datenblock in der Größe des zurückgegebenen Wertes an, der im Anschluss an den Schleifendurchlauf ausgelesen und weiterverarbeitet werden kann.

Analyseaspekt 1: Ressourceneffizienz

Bei der Betrachtung der Ressourceneffizienz kann der Speicherverbrauch zugunsten

ten der CPU-Auslastung vernachlässigt werden, da durch die beschriebene *Polling*-Methodik keine zusätzliche Speichernutzung zu verzeichnen ist. Die Auslastung der CPU ist durch den wiederkehrenden Schleifendurchlauf extrem hoch und somit nicht energieeffizient. Es ist jedoch zu beachten, dass die CPU-Auslastung nur dann von Relevanz ist, wenn diese zum Zeitpunkt des *Busy-Waitings* eine andere Aufgabe hätte übernehmen können. Der Einsatz der nicht-blockierenden Programmierung ist somit in deutlicher Abhängigkeit vom Gegenstand der Implementierung zu bewerten. Sie findet beispielsweise ihre Anwendung im *High Performance Computing*, da dort das Erzeugen eines *Interrupts* teurer ist, als das *Polling* auf ein Register.

```
1       bool a = false;
2       bool b = false;
3
4       while (a | b)
5       {
6           a = socket1.Available == 0 ? false : true;
7           b = socket2.Available == 0 ? false : true;
8       }
```

Auflistung 3.1: Polling auf zwei Sockets

Analyseaspekt 2: Quellcodestrukturierung

Der Quelltext wird beim Einsatz der *Busy-Waiting*-Mechanismen naturgemäß durch eine Vielzahl an Schleifen geprägt, die die zugrundeliegende Logik umschließen. Die Methodik zur Implementierung von komplexen Ereignissen eignet sich im Gegensatz zum vorangegangenen Modell sehr gut. Das komplexe Ereignis kann innerhalb der Abbruchbedingung einer Schleife definiert werden, in deren Rumpf die zugehörigen Methoden aufgerufen werden. Lauscht eine Applikation beispielsweise auf zwei unterschiedlichen *Sockets* und sieht die Applikationslogik vor, dass eine Aktion ausgeführt werden soll, sobald an einem der beiden *Sockets* Daten anliegen, so lässt sich das sehr elegant wie in Auflistung 3.1 darstellen. Bei jedem Durchlauf des Schleifenrumpfs wird geprüft, ob an einem der beiden *Sockets* Daten anliegen. Das Resultat wird in den beiden booleanschen Variablen a und b abgelegt, die in der Schleifenbedingung in Zeile 4 ausgewertet werden.

Analyseaspekt 3: Synchronisierung

Die Frage der *Thread*-Synchronisierung ist beim nicht-blockierenden Programmiermodell nicht zu stellen, da das Modell den auszuführenden Quelltext ausschließlich

sequentiell mit einem *Thread* abarbeitet. Der Einsatz von Synchronisierungsmechanismen ist somit unnötig.

3.2.3 Modell III: Asynchrone Programmierung

Das Modell der asynchronen Programmierung [113, 119] ist eines der effizientesten und zudem komplexesten Verfahren paralleler Programmierung. Der Entwickler stößt dabei einen Methodenaufruf an und übergibt diesem neben den für die Methode nötigen Parametern eine Referenz auf eine Rückrufmethode. Sobald das Ergebnis der aufgerufenen Methode vorliegt, ruft ein beliebiger *Thread* aus dem *Thread-Pool* des Betriebssystems den Funktionszeiger der Rückrufmethode auf. Dieser Aufruf dient als Information, dass ein Ergebnis anliegt. Das Ergebnis selbst wird nicht übergeben, sondern kann über den Aufruf einer zweiten Methode abgefragt werden.

Analyseaspekt 1: Ressourceneffizienz

Das Verfahren arbeitet sowohl Speicher- als auch Rechenzeiteffizient, da weder ein *Thread* blockiert und somit Speicher allokiert, noch Rechenzeit durch *Busy-Waiting*-Mechanismen verbraucht wird. Der Aufruf des Funktionszeigers der Rückrufmethode wird von einem *Thread* aus dem Pool vorgenommen und bietet somit die optimale Performanz, da das teure Erstellen eines *Threads* umgangen wird. Das Verfahren weist somit die optimale Ressourceneffizienz gepaart mit maximaler Performanz und Skalierbarkeit auf.

Analyseaspekt 2: Quellcodestrukturierung

Der Nachteil des äußerst effizienten Modells wird in der Darstellung des Quelltextes sichtbar. Der permanente und unumgängliche Einsatz von Funktionszeiger und Rückrufmethoden reißt die zu implementierende Applikationslogik auseinander und verteilt diese auf die einzelnen *Callback-Handler*. Die Aufsplittung des Quelltextes wird am einfachsten an einem relativ simplen Beispiel sichtbar. Eine Applikation wartet auf den Eingang von Daten auf *Socket1,* um im Anschluss Daten auf *Socket2* zu empfangen. Die aufgezeigte Logik wäre im Modell der blockierenden Programmierung durch einen Zweizeiler implementierbar, bei dem in Zeile 1 die Methode *Receive* auf *Socket1* und in Zeile 2 dieselbe Methode auf *Socket2* aufgerufen werden würde. Der Aufwand im asynchronen Modell ist erheblich größer und in Auflistung 3.2 dargestellt. Die asynchrone Variante erfordert für jeden Aufruf der *BeginReceive*-Methode einen Zeiger auf die entsprechende Rückrufmethode, die die weitergehende Logik implementiert.

```
1       public void Start()
2       {
3           socket1.BeginReceive(buffer, offset, size, SocketFlags.None,
                Socket1Callback, null);
4       }
5
6       private void Socket1Callback(IasyncResult result)
7       {
8           int count = socket1.EndReceive(result);
9           // ... Store data ...
10
11          socket2.BeginReceive(buffer, offset, size, SocketFlags.None,
                Socket2Callback, null);
12      }
13
14      private void Socket2Callback(IasyncResult result)
15      {
16          int count = socket2.EndReceive(result);
17          // ... Store data ...
18          // proceed with application logic
19      }
```

Auflistung 3.2: Asynchroner Zugriff aus zwei Socket-Instanzen

Die Darstellung von komplexen Ereignissen ist im Modell der asynchronen Programmierung möglich, erfordert aber aufgrund der wenig intuitiven Quelltextstruktur erheblichen Aufwand und bedingt zudem den permanenten Einsatz eines der bekannten Synchronisierungsmechanismen.

Analyseaspekt 3: Synchronisierung

Der Aspekt der Synchronisierung spielt beim Modell der asynchronen Programmierung eine vergleichsweise große Rolle. Da jeder Aufruf einer *Callback*-Methode im Kontext eines beliebigen *Threads* aus der Menge der *Thread-Pool-Threads* erfolgt, ist in jedem Fall der Einsatz eines Synchronisierungsmechanismus nötig, insofern auf eine geteilte Ressource zugegriffen wird. Die Definition geteilter Ressourcen beinhaltet nicht nur allokierte Hardwareressourcen sondern auch die Felder und Eigenschaften der umschließenden Klasse. Der Zugriff auf geteilte Ressourcen und die damit verbundene Synchronisierung ist somit eher der Standardfall als die Ausnahme. Dieses muss von Anbeginn der Implementierung beachtet werden, um Effekte der Nebenläufigkeit zu vermeiden.

3.2.4 Modell IV: Aktor-basierte Programmierung

Den Abschluss des Kapitels der Programmiermodelle bildet die Aktor-basierte Programmierung [3, 4, 25]. Das 1973 vorgestellte mathematische Modell der Aktoren [52] definiert einen Aktor als das kleinste, parallel beziehungsweise simultan auszuführende Konstrukt. Jeder Aktor trifft dabei ausschließlich lokale Entscheidungen. Ein globales Wissen über alle Aktoren des Gesamtsystems existiert nicht. Die Kommunikation der Aktoren verläuft ausschließlich über den Austausch von Nachrichten, die über ein Netzwerk oder andere unabhängige Kommunikationsmechanismen zugestellt werden können. Eine weitere Eigenschaft der Aktoren besagt, dass diese weitere Aktoren erstellen und selbst nach Verrichtung ihrer definierten Aufgabe terminieren können. Die Unabhängigkeit der einzelnen Aktoren wird nicht ausschließlich durch die Nachrichten-basierte Kommunikation, sondern durch die Forderung nach autarkem Speicher und einer eigenständigen Recheneinheit deutlich.

Das Aktorenmodell unterscheidet sich von den vorherig vorgestellten Modellen dahingehend, dass es als *Software-Pattern* zu betrachten ist und somit in keinem der verbreiteten Programmierumgebungen enthalten ist. Die Klassifikation und Analyse dieses Modells unterscheidet sich daher von den vorherigen Modellen und betrachtet in jeder der angegebenen Dimensionen eine optimale Implementierung des *Software-Patterns*.

Analyseaspekt 1: Ressourceneffizienz

Die Ressourceneffizienz des Aktor-Modells ist daher von der realen Implementierung des Konzeptes abhängig. Die geforderte Nachrichten-basierte Kommunikation kann auf einem beliebigen der drei vorangegangenen Programmiermodelle implementiert werden und bietet somit die aufgezeigten Vor- und Nachteile des jeweiligen Konzeptes. Eine reale Implementierung des Aktorenmodells kann somit bei der Auswahl des asynchronen Programmiermodells sehr ressourceneffizient gestaltet werden.

Analyseaspekt 2: Quellcodestrukturierung

Die Auswahl des Programmiermodells wirkt sich ebenfalls auf die Strukturierung des Quelltextes und die Erkennung von komplexen Ereignissen aus. Bei der Umsetzung des Aktorenmodells kann somit durch die Wahl eines geeigneten Programmiermodells eine optimale Strukturierung des Quelltextes erreicht werden. Es ist jedoch zu beachten, dass die Zeile des ersten und zweiten Analyseaspektes im Bezug auf die drei vorgestellten Programmiermodelle der blockierenden, nicht-blockierenden

und asynchronen Programmierung konträr zu betrachten ist. Eine optimale Quellcodestrukturierung steht beispielsweise im Gegensatz zur Auswahl des asynchronen Modells zur Optimierung der Ressourceneffizienz.

Analyseaspekt 3: Synchronisierung

Der dritte und letzte Analysepunkt behandelt die Synchronisierungseigenschaften des Aktor-Modells. Jeder Aktor verfügt entsprechend dem Modell über eine eigene Recheneinheit und zudem über eigenen Speicher. Die Ausführung innerhalb des Aktors erfolgt rein sequentiell. Die Frage nach der Synchronisierung entfällt, insofern sichergestellt ist, dass die Zugriffe auf die Nachrichtenwarteschlange *Thread*-sicher erfolgen.

3.3 Anforderungsanalyse

Die Analyse der in Kapitel 3.2 vorgestellten Programmiermodelle hat sowohl die Stärken als auch die Schwächen der ersten drei Modelle in allen drei Analysedimensionen offenbart. Die in Tabelle 3.2 zusammengefassten Ergebnisse zeigen, dass keines der Modelle mit realer Implementierung in allen Aspekten der Analyse Stärken aufweisen kann. Die Aussagekraft der Analyse über das *Software-Pattern* der aktorbasierten Programmierung verdeutlicht, dass das Verfahren in jedem Analyseaspekt optimale Ergebnisse liefern könnte, insofern bei jedem Analyseaspekt das optimale der drei vorgestellten Programmiermodelle zugrunde liegt. Die Auswahl des aktorbasierten Verfahrens als optimales Modell ist jedoch aufgrund der konträre Ziele der für die unterschiedlichen Analyseaspekte gewählten optimalen Programmiermodelle nicht möglich.

Das eindeutige Ergebnis der Programmiermodellanalyse erfordert die Entwicklung eines neuen Programmiermodells, das die Stärken der vorgestellten Modelle dahingehend kombiniert, dass das neue Modell den Analyseaspekten in allen Dimensionen entspricht. Das neue Modell muss sich daher im ersten Analyseaspekt an der effektiven und effizienten Nutzung von CPU-Zyklen sowie Speicher messen lassen, um Grundlage für skalierende und hoch performante Systeme zu bilden. Zweitens muss das Modell mit der strukturierten Darstellung des Quelltextes, sowie mit der einfachen Definition von komplexen Ereignissen überzeugen, wie dieses in Auflistung 3.1 dargestellt ist. Die Anforderung, die aus dem dritten Analyseaspekt abgeleitet wird, sieht vor, dass das neue Modell keinerlei explizite Synchronisierung erfordern darf, insofern die Grenzen des Modells nicht überschritten werden.

Modell	I	II	III	IV
CPU Auslastung				
Teure Thread Erzeugung	ja	nein	nein	nein
Schedulingaufwand	ja	nein	nein	nein
Pollingaufwand	nein	ja	nein	nein
Speichernutzung	hoch	gering	gering	gering
Skalierbar	nein	nein	ja	ja
Performanz	ja/nein	nein	ja	ja
Strukturierter Quelltext				
Zustandsmaschinen ähnlich	ja	nein	nein	ja
Callback-Handler	nein/ja	nein	ja	nein
Polling-Schleifen	nein	ja	nein	nein
Ereigniserkennung				
Erfordert Modellabweichung	ja	nein	nein	nein
Synchronisierung erforderlich	ja/nein	nein	ja	nein

Tabelle 3.2: Zusammenfassung der Programmiermodellanalyse - Die Angaben des Aktorenmodells basieren auf dem jeweilig optimalen Programmiermodell für den betrachteten Analyseaspekt.

Neben den aus der Programmiermodellanalyse abgeleiteten Anforderungen an das neu zu entwickelnde Modell sind noch drei weitere Forderungen aus der klassischen Softwareentwicklung zu stellen. Die Entwicklung von Applikationen mit dem neuen Modell darf erstens keinerlei zusätzlichen Implementierungsaufwand bedeuten und muss flexibel in unterschiedlichen Applikationstypen eingesetzt werden können. Zweitens muss das Modell eine vollständige Unterstützung durch das zugrundeliegende Framework und einen Standard Compiler bieten. Abschließend wird die Forderung nach möglichst einfachen *Debugging*-Mechanismen aufgeworfen, die durch das neue Modell erfüllt werden soll.

3.4 Das Gears4Net Programmiermodell

Die Konzeption eines neuen Programmiermodells für massiv parallele Applikationen müsste die bestehenden Modelle dahingehend kombinieren, dass ausschließlich die Stärken der Systeme zum Tragen kommen, ohne dass die jeweiligen Schwächen in das neue Modell einfließen. Das neue Modell wird zukünftig mit dem Namen *Gears4Net* [110] bezeichnet. Es kombiniert die unterschiedlichen Teilkonzepte der vier

Modelle und vereinigt somit die maximale Ressourceneffizienz mit optimaler Quellcodestrukturierung und Organisation sowie synchronisierungsfreier Programmierung, ohne Änderungen am zugrundeliegenden Betriebssystem zu fordern.

3.4.1 Grundidee des Programmiermodells Gears4Net

Die grundlegende Idee des *Gears4Net*-Programmiermodells ist durch die Konzeption des von Hewitt, Bishop und Steiger 1973 vorgestellten Aktorenmodells [52] inspiriert worden, da dieses Modell ohne störende Synchronisierungsmechanismen auskommt. Das Aktorenmodell ermöglicht die synchronisierungsfreie, parallele Programmierung, indem es autarke, ausschließlich über Nachrichten kommunizierende Aktoren bereitstellt, die jeweils über eine eigene Recheneinheit mit nicht-geteiltem Speicher verfügen.

Überführt man die Konzepte des Aktorenmodells aus dem Jahre 1973 auf aktuelle Softwareentwicklungsmethoden, so lassen sich Aktoren als aktive Objekte betrachten, die als Container beliebiger Funktionalität dienen und über Nachrichtenwarteschlangen miteinander kommunizieren. Die Transformation des Ausführungskonzeptes auf moderne Rechnerarchitekturen erfordert jedoch eine größere Anpassung, da ab einer relativ geringen Anzahl unabhängiger Objekte nicht jedem Objekt eine eigene Recheneinheit in Formen einer CPU oder eines *Threads* zugewiesen werden kann. Es ist daher unabdingbar, die Ausführungsumgebung dahingehend zu konzipieren, dass ein ausführender *Thread* mehreren unabhängigen Objekten zugewiesen werden kann, aber jedes Objekt zu einem Zeitpunkt ausschließlich von diesem einen zugewiesenen *Thread* ausgeführt wird. Dies stellt sicher, dass zu jedem Zeitpunkt höchstens ein *Thread* das unabhängige Objekt ausführt und keine Effekte der Nebenläufigkeit auftreten können, da kein zweiter *Thread* in die Ausführung involviert ist.

Die Zuweisung eines *Threads* an mehrere Objekte erfordert die Einschränkung, dass ausschließlich nicht-blockierende Funktionsaufrufe innerhalb der unabhängigen Objekte getätigt werden dürfen, da ein blockierender Aufruf die Blockade des ausführenden *Threads* nach sich ziehen würde und damit die Ausführung aller dem *Thread* zugewiesenen Objekte unterbindet.

Die Auswahl geeigneter Konzepte zur Erfüllung der Kriterien der ersten beiden Analysedimensionen ist signifikant für die optimale Konzeption des neuen *Gears4Net*-Programmiermodels. In Bezug auf die effiziente Ressourcennutzung bildet das Modell der asynchronen Programmierung den Primus in der ersten Analysedimension.

Die Auswirkungen des Modells der asynchronen Programmierung schlagen, wie in Tabelle 3.2 aufgezeigt, direkt auf die Struktur des Quelltextes durch und zwingen den Entwickler, mit einer Vielzahl an *Event-Handlern* und Rückrufmethoden umzugehen. Der Umgang mit diesem Modell ist im Gegensatz zur blockierenden Programmierung, bei dem Zustandsmaschinen-ähnlich Befehl nach Befehl abgearbeitet wird, wenig intuitiv und wirft den Wunsch nach einem Modell auf, das asynchron arbeitet und dabei nicht auf den Komfort der Zustandsmaschinen-ähnlichen Programmierung verzichtet. Das *Gears4Net*-Modell erreicht diesen Spagat durch das Zusammenspiel der Ausführungsumgebung mit unterbrechbaren Methoden, in denen Wartebedingungen spezifiziert werden können, die einem blockierenden Systemaufruf ähneln. Die Ausführungsumgebung führt eine unterbrechbare Methode solange aus, bis eine Wartebedingung erreicht ist. Insofern die Bedingung instantan erfüllt ist, wird mit der Ausführung der Methode fortgefahren. Andernfalls entzieht die Ausführungsumgebung der aktuellen Methode den rechenbereiten *Thread* und setzt die Ausführung der Methode erst dann wieder fort, wenn die Wartebedingung erfüllt worden ist.

Das vorgestellte Konzept wird nochmals anhand des Pseudocode Beispiels in Auflistung 3.3 verdeutlicht. Die unterbrechbare Funktion *Execute* ruft die asynchron arbeitende und sofort zurückkehrende Methode *SendRequestAsync* in Zeile 3 auf und wartet anschließend auf den Eingang des Ergebnisses durch den Aufruf der Wartebedingung *Wait(asyncResult)* in Zeile 4. Die Ausführungsumgebung prüft im Anschluss an den Aufruf der *Wait*-Methode, ob das Ergebnis des asynchronen Methodenaufrufs eingegangen ist. Ist dies der Fall, wird das Ergebnis über die *Print*-Methode ausgegeben. Insofern das Ergebnis nicht vorliegt, entzieht die Ausführungsumgebung der *Execute*-Methode den ausführenden *Thread* und weist diesen erst dann wieder zu, wenn das Ergebnis anliegt und mit der Ausführung der *Print*-Methode fortgesetzt werden kann.

```
1    function Execute()
2    {
3        SendRequestAsync()
4        Wait(asyncResult)
5        Print(asyncResult)
6    }
```

Auflistung 3.3: Unterbrechbare Methoden und Wartebedingungen

Die Realisierung der Wartebedingungen steht in direktem Zusammenhang mit der Nachrichten-basierten Kommunikation des Aktorenmodells. Die Spezifikation einer Wartebedingung ist als Äquivalent mit dem Eingang einer dedizierten Nachricht zu

betrachten. Bezogen auf das gegebene Beispiel bedeutet dies, dass die asynchron aufgerufene *SendRequestAsync*-Methode bei Anliegen des erwarteten Resultates automatisiert eine Nachricht erzeugt und diese in die Nachrichtenwarteschlange einstellt, die wiederum durch die in Zeile 4 des Beispiels dargestellte Wartebedingung erfasst wird.

Die Konzeption des *Gears4Net*-Modells wird durch die Modellierung elementarer Wartebedingungen abgeschlossen, die neben der Definition eines singulären Ereignisses auch die Spezifikation von beliebig komplexen Wartebedingungen erlauben und sich aus der Disjunktion und Konjunktion singulärer Ereignisse zusammensetzen und damit die letzte Anforderung an ein Programmiermodell für massiv parallele Anwendungen erfüllen.

3.4.2 Die Gears4Net Modellarchitektur

Vor der Implementierung der *Gears4Net*-Idee steht die Ausarbeitung einer geeigneten Softwarearchitektur, die die einzelnen Teilkonzepte der Programmiermodellidee in Architekturbausteine umwandelt und diese zu einer konsistenten Gesamtarchitektur zusammenführt. Die *Gears4Net*-Architektur extrahiert dazu drei Grundbausteine aus dem Aktorenmodell, die in die Gesamtarchitektur einfließen. Zu diesen zählen der Aktor, der als *Protocol* bezeichnet in die Architektur überführt wird, sowie die für den asynchronen Nachrichtenaustausch notwendige *Message-Queue*. Der dritte und letzte Grundbaustein des Aktorenmodells ist die Ausführungsumgebung, die jedem Aktor einen eigenen *Micro*-Prozessor mit autarken Speicher zusichert. Dieser Baustein wird in Form von *Schedulern*, die oberhalb des Betriebssystems angesiedelt werden, in die Architektur eingeführt. Neben den drei Grundbausteinen aus dem Aktorenmodell fließen drei weitere Konzepte zur Optimierung der Quellcodestruktur in die Architektur ein. Die Umsetzung der unterbrechbaren Methoden erfolgt über die Einführung von *State-Machines*, die beliebig komplexe Wartebedingungen in Form von verschachtelten *Receiver*-Objekten enthalten können. Abgeschlossen wird das Architekturschaubild mit der Einführung der *Signal*-Bausteine, die das Anzeigen von Ereignissen vereinfachen.

Das Architekturdiagramm in Abbildung 3.4 zeigt das Zusammenspiel der sechs Elementarbausteine der *Gears4Net*-Architektur und präsentiert dies in einem klassischem Schichtenmodell. Die Ausführungsschicht des *Gears4Net*-Programmiermodells bildet die erste Architekturebene. Die lose Kopplung der *Sche-*

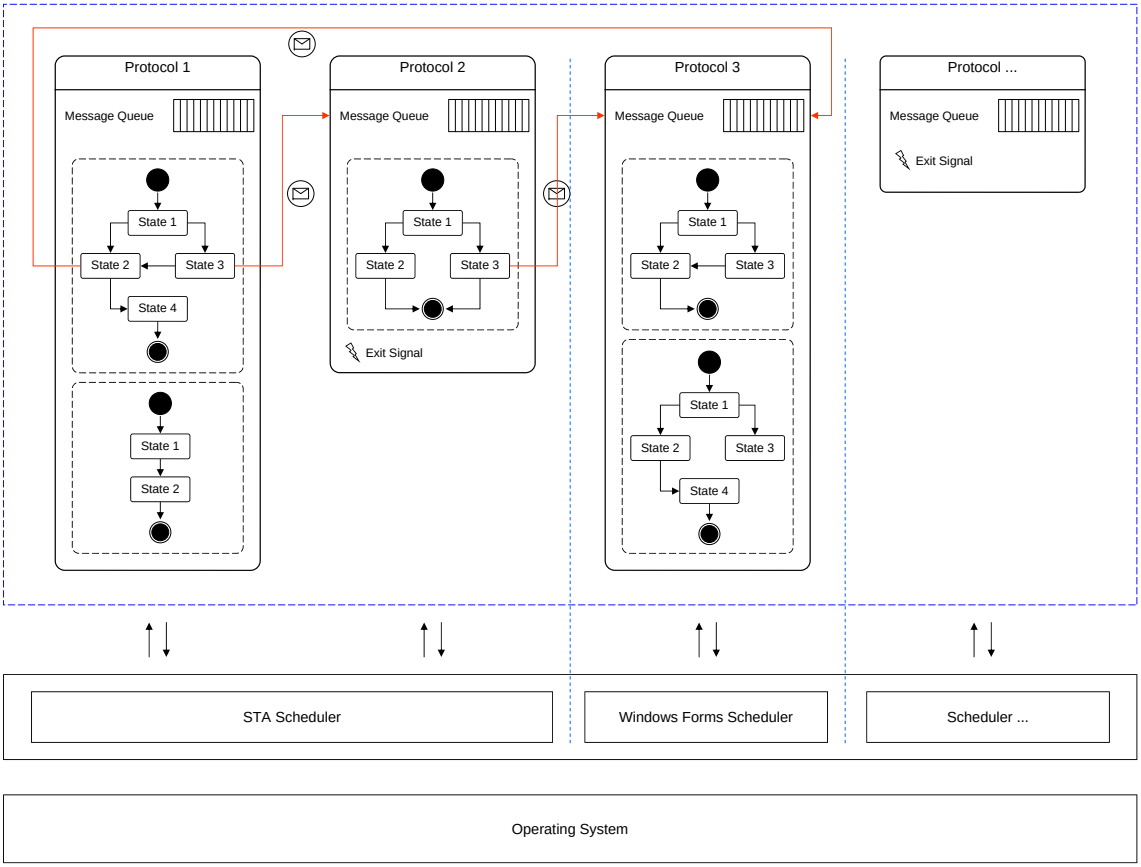

Abbildung 3.4: Übersicht des Gears4Net Architekturdiagramms

duler mit dem darunterliegenden Betriebssystem garantiert dabei die geforderte Betriebssystemunabhängigkeit.

Die oberste Schicht wird durch die *Protocol*-Instanzen gebildet, die jeweils über eine eigene *Message-Queue* verfügen, die zur internen und externen Kommunikation konzipiert wurde. Die Quellcodestrukturierung des blockierenden Programmiermodells erhält durch die Bereitstellung einer initialen *State-Machine* pro Instanz Einzug, von der aus weitere Untermaschinen erzeugt werden können. Die Zustände, der im Architekturdiagramm durch gestrichelte Rechtecke dargestellten Maschinen, werden durch *Receiver*-Wartebedingungen abgebildet, die auch die Definition komplexer Konstrukte zulassen. Abgeschlossen wird das Diagramm mit den Konzept der *Signale*, die in den *Protocol*-Instanzen 2 und 4 dargestellt sind und ausschließlich über den Eintritt eines Ereignisses informieren, ohne dabei Nutzdaten zu transportieren.

Die sechs Elementarbausteine der *Gears4Net*-Architektur werden in den folgenden Abschnitten detailliert beleuchtet und beschrieben. Die Beschreibungsreihenfolge der Bausteine orientiert sich dabei am Aufbau des Architekturdiagramms und verläuft

von den unteren zu den oberen Schichten sowie von außen nach innen innerhalb der zu erläuternden Schicht.

Scheduler

Die *Scheduling*-Schicht des *Gears4Net*-Modells setzt direkt auf dem Betriebssystem auf und erlaubt die Implementierung von verschiedenartigen *Scheduling*-Verfahren und Algorithmen. Die Implementierung verschiedenartiger Verfahren beziehungsweise deren Auswahl ist abhängig von der zugrundeliegenden Rechnerarchitektur und den Zielen der Applikation. Beispielsweise ist es sinnvoller, auf einer Einprozessormaschine einen anderen *Scheduling*-Algorithmus zu verwenden als auf einer Mehrprozessormaschine, die einen höheren Grad an paralleler Verarbeitung zulässt.

Die Zuweisung von *Protocol*-Instanzen und *Scheduler*-Objekten ist nicht bijektiv. Einem *Scheduler* können beliebig viele *Protocol*-Instanzen zugewiesen werden, eine *Protocol*-Instanz verfügt jedoch ausschließlich über einen *Scheduler*. Die Arbeitsweise eines *Gears4Net-Schedulers* unterscheidet sich konzeptionell wenig von *Schedulern* bestehender Betriebssysteme mit dem einzigen Unterschied, dass im Fall der *Gears4Net*-Variante keine *Threads* sondern *Protocol*-Instanzen den *Schedulding*-Algorithmen unterworfen sind. Ein *Scheduler* beziehungsweise dessen ausführender *Thread* bleibt solange aktiv, bis dieser nicht mehr über rechenbereite *Protocol*-Instanzen verfügt. Eine *Protocol*-Instanz wird als rechenbereit bezeichnet, wenn die Nachrichtenwarteschlange der Instanz Einträge enthält, die von der Instanz verarbeitet werden können. Verfügt ein *Scheduler* nicht mehr über rechenbereite *Protocol*-Instanzen, so wird der ausführende *Thread* des *Schedulers* in einen schlafenden Zustand versetzt und erst wieder aktiviert, wenn einer *Protocol*-Instanz eine Nachricht zugestellt wird. Dieses Verhalten ermöglicht dem *Scheduler* des Betriebssystems, aktuell inaktive *Gears4Net-Scheduler* im *Scheduling*-Prozess zu ignorieren und damit die maximale Performanz und Effizienz zu erreichen.

Protocol

Das Konzept der *Protocol*-Instanzen resultiert aus der Überführung der klassischen Aktoren in die Welt der objektorientierten Programmierung. Ein *Protocol* versteht sich somit als kleinste, parallel auszuführende Einheit, da jegliche Ausführung innerhalb einer *Protocol*-Instanz sequentiell erfolgt. Dieses Verständnis erlaubt, Parallelität als Designziel in den Prozess der Applikationsentwicklung aufzunehmen und die Architektur von massiv parallelen Anwendungen dahingehend abzustimmen, dass viele unabhängige und transparent verteilbare *Protocol*-Instanzen entstehen können.

Eine *Protocol*-Instanz ist als ein abhängigkeitsfreier Container beliebiger Funktionalität zu betrachten, dessen in- und externe Kommunikation über die Nachrichtenwarteschlange der Instanz verläuft und dessen Ausführung von einem *Scheduler* des *Gears4Net*-Modells übernommen wird. Mit der Ausführung einer *Protocol*-Instanz wird die Verarbeitung einer Nachricht aus der *Message-Queue* der *Protocol*-Instanz bezeichnet. Befindet sich eine Instanz in einem rechenbereiten Zustand, so wird das *Protocol* vom *Scheduler* aufgefordert, die erste Nachricht der Warteschlange zu verarbeiten und anschließend die Kontrolle an den *Scheduler* zurückzugeben. Die Nachrichtenverarbeitung erfolgt innerhalb einer der in der Instanz erzeugten Zustandsmaschinen. Jede *Protocol*-Instanz stellt dazu eine initiale *State-Maschine* bereit, von der beliebig viele Untermaschinen abgespalten werden können. Eine Nachricht wird genau dann von einer Zustandsmaschine verarbeitet, wenn die Spezifikation einer Wartebedingung der eingehenden Nachricht entspricht, die in Form von mehr oder weniger komplexen *Receiver*-Ausdrücken definiert werden können.

Message-Queue

Die Nachrichtenwarteschlange des *Gears4Net*-Modells wird durch den *Message-Queue*-Architekturbaustein abgebildet. Die *Message-Queue* verfügt über die typischen *Enqueue*- und *Dequeue*-Mechanismen zur Aufnahme und Abgabe von Nachrichtenobjekten und erweitert diese standardisierte Funktionalität mit Konzepten der Flusskontrolle und der *Receiver*-Wartebedingungen.

Die Wirkung der Flusskontrolle zeigt sich beim Einstellen einer Nachricht in die Warteschlange. Sollte die maximale Kapazität der *Queue* erreicht sein, so wird die Nachricht verworfen, insofern es sich nicht um eine systemrelevante Information handelt. Diese Architekturentscheidung versucht das mögliche Volllaufen der Warteschlangen und des Arbeitsspeichers beispielsweise durch *DoS*-Angriffe auf das System zu verhindern.

Die Konzeption der *Receiver*-Wartebedingungen erlaubt die Benachrichtigung über den Eingang einer spezifizierten Nachricht in der Warteschlange. Somit kann der Entwickler, der beispielsweise auf den Eingang einer Nachricht mit einem festgelegten Identifizierer wartet, über den Eingang informiert werden.

State-Machine

Die Beschreibung der drei vorangegangenen Konzepte, die nahezu direkt aus dem Aktorenmodell abgeleitet wurden, dienen der Ausführung von autarken, parallelen Einheiten und deren Kommunikation. Die Architekturbausteine der *State-Machines*

sowie der im folgenden Abschnitt beschriebenen *Receiver*-Wartebedingungen konzentrieren sich hingegen auf eine möglichst optimale Strukturierung des Quelltextes und damit auf höchst effiziente Entwicklungsmethoden.

Eine *State-Machine*, im Folgenden auch als Zustandsmaschine oder Automat bezeichnet, wird als eine unterbrechbare Methode aufgefasst, der eine zu verarbeitende Nachricht als Parameter übergeben wird. Die Methode wird samt der übergebenen Nachricht genau solange ausgeführt, bis der *Program-Counter* auf eine *Receiver*-Wartebedingung zeigt. Sobald die Bedingung des spezifizierten *Receiver*-Objektes erfüllt ist, wird die bereits übergebene alte Nachricht durch die neue ersetzt und die Ausführung der Methode an dem Punkt fortgesetzt, an dem diese unterbrochen wurde.

Jede *Protocol*-Instanz verfügt über eine initiale Zustandsmaschine, von der aus Strukturierungsgründen weitere Maschinen abgespalten werden können. Die Zustellung der eingehenden Nachrichten an die relevanten Untermaschinen erfolgt völlig transparent und ohne Eingriff des Entwicklers. Ein beliebig großes Zustandsdiagramm inklusive der Zustandsüberführungsfunktionen kann somit in kleine, verständliche Quellcodeblöcke unterteilt werden und neben der Les- und Wartbarkeit vor allem das *Debugging* und Testen des Quelltextes vereinfachen.

Das vorgestellte Konzept der *State-Machine* entspricht vollständig der Semantik des theoretischen Automatenkonzeptes, da jedes der in der Definition eines Automaten enthaltenen Tupelelemente im Konzept der unterbrechbaren Methoden wiederzufinden ist. Ein klassischer Automat ist als Fünf-Tupel, bestehend aus Eingabealphabet, Zustandsmenge, Überführungsfunktion, Startzustand und Endzustand, definiert. Das Eingabealphabet entspricht den Nachrichten, die der unterbrechbaren Methode übergeben werden, wobei Nachrichten, die nicht im Alphabet enthalten sind, ignoriert werden. Die Menge der Zustände ist durch die Anzahl der *Receiver*-Wartebedingungen spezifiziert. Eine *State-Machine* verbleibt solange in ihrem Zustand, bis eine eingehende Nachricht die Überführung in den nächsten Zustand veranlasst. Der Quellcode zwischen zwei *Receiver*-Wartebedingungen ist somit als Zustandsüberführungsfunktion zu betrachten und erfüllt damit die Anforderung der dritten Tupeleigenschaft. Die Entsprechungen von Anfangs- und Endzustand sind ebenfalls direkt aus dem Konzept der unterbrechbaren Methoden ersichtlich. Der Anfangszustand ist durch den Eintritt in den Methodenrumpf definiert, der Endzustand beziehungsweise die Terminierung des Automaten durch Verlassen desselben.

Abschließend ist anzumerken, dass die Terminierung eines Untermaschine ausschließlich Einfluss auf den Automaten selbst hat, während die Terminierung der initia-

len Zustandsmaschine die Terminierung der gesamten *Protocol*-Instanz nach sich zieht.

Receiver

Die *Receiver*-Wartebedingungen sind der zweite Architekturbaustein, um die Ziele der Zustandsmaschinen-ähnlichen Programmierung in Verbindung mit der Definition komplexer Ausdrücke zu ermöglichen. Als Bestandteil einer unterbrechbaren Methode, sind die *Receiver*-Wartebedingungen, die einen Automaten in einem definierten Zustand halten, das letzte Glied in der Kette der Nachrichtenverarbeitung. Sobald ein *Scheduler* die Verarbeitung einer in der *Message-Queue* einer *Protocol*-Instanz enthaltenen Nachricht anstößt, wird diese an die initiale *State-Machine* übergeben und von dort aus an die abgespaltenen Subautomaten weiter transferiert. Jede Zustandsmaschine leitet die entgegengenommene Nachricht an die aktive Wartebedingung weiter, die den aktuellen Zustand repräsentiert. Sollte die Nachricht die aufgestellte Bedingung erfüllen, setzt instantan die Überführungsfunktion in den nächsten Zustand ein. Andernfalls wird die Nachricht nicht weiter behandelt und von der aktuellen Zustandsmaschine verworfen.

Der zweite Aufgabenbereich der *Receiver*-Wartebedingung obliegt der Erkennung komplexer Ereignisse. Ein solche komplexe Struktur kann beispielsweise entstehen, wenn auf den Eingang der Nachrichten a und b oder der Nachricht c gewartet werden muss, um eine Zustandstransformation durchzuführen. Das *Receiver*-Konzept erlaubt es daher, verschiedenartige Wartebedingungen ineinander zu verschachteln, um komplexe, baumartige *Receiver*-Strukturen zu erhalten, die automatisiert evaluiert werden können.

Signals

Die in- und externe Kommunikation einer *Protocol*-Instanz erfolgt über Nachrichten, die in die jeweilige Warteschlange der Zielinstanz eingestellt werden. Der Entwickler definiert dazu für jede Nachrichtenklasse einen speziellen Nachrichtentyp, der mit den jeweils relevanten Nutzdaten versehen und dem Empfänger zugestellt wird. Liegen beispielsweise an einem Netzwerk-*Socket* Daten an, kann der empfangene Datenblock in eine *DataReceivedMsg* verpackt und der *Protocol*-Instanz zugestellt werden, die auf den Eingang genau dieser Nachricht gewartet hat.

Die Nachrichten-basierte Kommunikation dient im vorgestellten Beispiel ausschließlich dem Zustellen der enthaltenen Nutzdaten und ermöglicht somit deren anschließende Weiterverarbeitung. Die Semantik einer Nachricht kann jedoch noch auf ei-

ne zweite Weise definiert werden, indem das Eintreffen der Nachricht bereits die erwartete Information ist. Insofern eine Nachrichtenklasse ohne enthaltene Nutzdaten auskommt, würde dies die Definition von Nachrichtentypen mit einem leeren Rumpf erfordern. Um die redundante Definition von inhaltsleeren Nachrichtentypen zu umgehen, führt das *Gears4Net*-Modell den sechsten Architekturbaustein, die *Signal*-Objekte, ein und bietet dem Entwickler somit weiteren Komfort.

Ein *Signal* ist ausschließlich für die Anzeige eines Ereignisses zuständig und beinhaltet keinerlei Nutzdaten. Jede *Signal*-Instanz erzeugt automatisch eine Nachricht, sobald das Ereignis eingetreten ist und stellt diese in die Warteschlange ein. Während das Warten auf den Eintritt eines *Signals* analog zum Eingang einer Nachricht über die Definition einer *Receiver*-Wartebedingung erfolgt, bedarf das Emittieren eines *Signals* lediglich des Aufrufs einer Methode auf der Instanz des auszulösenden *Signals*.

3.5 Plattform- und Sprachwahl

Die im vorangegangenen Kapitel beschriebene Architektur des *Gears4Net*-Programmiermodells fordert eine betriebssystemunabhängige Implementierung, die auf einer praxisrelevanten und weit verbreiteten Laufzeitumgebung beziehungsweise Programmiersprache aufsetzt. Für die folgende Analyse werden daher das *Microsoft .NET Framework 3.5* mit der Programmiersprache C#, das *Java Environment* in Version 1.6, sowie die Sprachen *Python* und *C++* herangezogen.

Die für die Analyse relevanten Kriterien können direkt aus der Architekturbeschreibung übernommen werden. Die Anforderung an eine geeignete Laufzeitumgebung beziehungsweise eine geeignete Programmiersprache beinhalten neben der Implementierbarkeit unterbrechbarer Methoden auch die Bereitstellung eines asynchronen Programmiermodells sowie die sehr weiche Anforderung an die sprachliche Eleganz. Letztere drückt sich beispielsweise in der intuitiven Darstellung von komplexen Wartebedingungen aus, die durch das Konzept des Überladens von Operatoren ermöglicht werden.

Das erste Kriterium des Anforderungskataloges, die Implementierbarkeit unterbrechbarer Methoden, stellt die größte Herausforderung an die Plattform, da dieses Kriterium von keinem der vier vorgeschlagenen Plattformen und Sprachen nativ unterstützt wird. Ein möglicher und zudem sehr effizienter Lösungsweg wird von [115] bei der Implementierung der *Concurrency and Coordination Runtime (CCR)* aufgezeigt.

Plattform	C#	Java	Python	C++
Iteratoren	Ja	Nein	Ja	Nein
Operatorüberladung	Ja	Nein	Ja	Ja
Asynchrones Programmiermodell	Ja	Ja/Nein	Ja	Ja

Tabelle 3.3: Entwicklungsplattform- und Spracheigenschaften

Der Ansatz bedient sich des aus *Python* bekannten Iteratorenkonzeptes und betrachtet unterbrechbare Methoden als Iteratoren. Eine solche unterbrechbare Methode ist genau dann vollständig ausgeführt, wenn der Iterator vollständig durchlaufen wurde. Jeder Iterationsschritt [7] trägt daher zur Gesamtausführung des Iterators bei und kann somit als Methodenunterbrechung angesehen werden.

Die Anforderungskriterien werden somit zugunsten des Iteratorenkonzeptes modifiziert und in Tabelle 3.3 dargestellt. Das Analyseergebnis offenbart, dass sich sowohl das *Microsoft .NET Framework 3.5* mit der Sprache *C#* als auch die Programmiersprache *Python* für die Implementierung des *Gears4Net*-Programmiermodells eignen. Die endgültige Auswahl zugunsten des *Microsoft .NET Frameworks* erfolgte aufgrund des Verbreitungsgrades und der Tatsache, dass mit der Programmiersprache *Iron Python* [74] eine Sprache aus der *Python*-Familie bereitgestellt wurde, die auf dem *Microsoft .NET Framework* ausgeführt werden kann.

3.6 Implementierung des Gears4Net Frameworks

Das folgende Kapitel beschreibt die Implementierung des Gears4Net Frameworks, das in der Programmiersprache C# entwickelt wurde und auf der Version 3.5 der Microsoft .NET Klassenbibliothek basiert. Gears4Net selbst besteht aus insgesamt 66 Datentypen, die sich aus 55 Klassen, einer Schnittstelle, acht Funktionszeigern und zwei Enumerationen zusammensetzen. Die Beschreibung der Bibliothek erfolgt anhand eines *Buttom-Up*-Ansatzes und unterteilt das Framework in die Bestandteile *Message-Queues*, *Signals*, *Receiver*, *State-Machines* und *Protocols* sowie den Teilbereich *Scheduler*. Jeder der sechs Teilaspekte wird im Folgenden separat vorgestellt und beschrieben. Die Interaktionen zwischen den einzelnen Teilbereichen werden durch Referenzen kenntlich gemacht um das Verständnis des Zusammenspiels der unterschiedlichen Komponenten zu verdeutlichen. Die Beschreibung des Frameworks folgt der Reihenfolge der aufgelisteten Bestandteile und beginnt insofern mit den Erläuterungen der *Message-Queue*-Klassen.

3.6.1 Message-Queues

Die asynchrone, nachrichtenbasierte Kommunikation der Gears4Net Architektur wird mittels spezialisierter *MessageQueues* abgebildet. Das Gears4Net Framework stellt dazu die beiden Klassen *MessageQueue<T>* und *BroadcastQueue*-Klasse bereit, die Nachrichtentypen verarbeiten können, welche die Schnittstelle *IMessage* implementieren. Die gemeinsamen Anforderungen der beiden *MessageQueue*-Klassen werden in der abstrakten Basisklasse *AbstractMessageQueue* bereit gestellt, die im Folgenden vorgestellt wird.

AbstractMessageQueue Basisklasse

Das Nachrichtenverarbeitungskonzept des Gears4Net Frameworks stellt drei Anforderungen an die zu implementierende Warteschlangen-Basisklasse. Die erste Forderung bedingt das *Thread*-sichere Hinzufügen und Entnehmen von Nachrichten aus der Warteschlange zur Vermeidung von *Race Conditions*. Zweitens ist die maximale Kapazität der *MessageQueue* konfigurierbar zu beschränken und abschließend muss der Forderung nachgekommen werden, dass mittels der in Kapitel 3.6.3 vorgestellten *Receiver* auf den Eingang einer Nachricht in die Schlange gewartet werden kann.

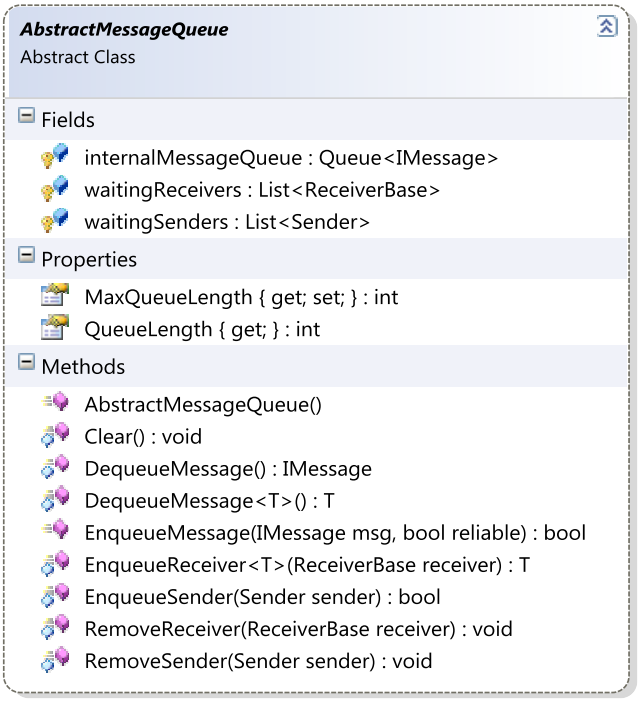

Abbildung 3.5: Klassendiagramm der AbstractMessageQueue-Basisklasse

Der abstrakte Datentyp *AbstractMessageQueue*, dessen Klassendiagramm in Abbildung 3.5 dargestellt ist, implementiert die aufgezeigten Anforderungen. Die *Abstract-*

MessageQueue-Klasse verwendet dazu die generische *Queue*-Datenstruktur aus dem *Namespace System.Collections.Generic* des Microsoft .NET Frameworks als interne Datenstruktur. Diese ist für die Verwaltung der eingehenden Nachrichten des Typs *IMessage* zuständig und wird dahingehend gekapselt, dass auf sie nur über *Thread*-sichere Methoden zugegriffen werden kann.

Die Funktionalität der *AbstractMessageQueue*-Klasse lässt sich in drei Funktionsblöcke unterteilen. Der erste Baustein implementiert das Hinzufügen und Herausnehmen von Nachrichten aus der Warteschlange. Diese Funktionalität wird von den Methoden *EnqueueMessage*, *DequeueMessage* sowie deren generischer Variante *DequeueMessage<T>* implementiert. Sollte beim Hinzufügen einer Nachricht der Füllstand der Warteschlange den über die Eigenschaft *MaxQueueLength* festgelegten Maximalwert überschreiten, greifen die in der Methode *EnqueueMessage* implementierte Flusskontrolle. Der zweite Funktionsbaustein implementiert das Konzept der *Sender*-Wartebedingungen. Diese sind dann von Bedeutung, wenn der Absender einer Nachricht darüber informiert werden möchte, wann diese in die *MessageQueue* eingestellt werden. Für die Verwaltung der *Sender* stehen im zweiten Funktionsbaustein die Methoden *EnqueueSender* und *RemoveSender* bereit. Der dritte und letzte Funktionsblock behandelt den Umgang mit *Receivern*, die auf den Eingang von Nachrichten in der *MessageQueue* warten. Diese Funktionalität wird durch die Methoden *EnqueueReceiver<T>* und *RemoveReceiver* abgebildet.

Die Semantik der vorgestellten Methoden folgt mit Ausnahme der generischen *DequeueMessage<T>*-Funktion dem intuitiven Verständnis, das durch die Methodenbezeichnungen vorgegeben ist. Die Implementierung der generischen *DequeueMessage<T>*-Methode, die in Auflistung 3.4 aufgezeigt ist, verfügt über den Typisierungsparameter *T*, über den bei Aufruf der Methode spezifiziert werden kann, welcher Typ Nachricht aus der Warteschlange entnommen werden soll. Befindet sich keine Nachricht in der *internalMessageQueue* oder entspricht das vorderste Element nicht vom Typisierungsparameter, so liefert die Methode den Rückgabewert *default(T)*. Sollte das vorderste Elemente der Warteschlange dem geforderten Typ entsprechen, so wird dieses wie in Zeile 9 beschrieben aus der *internalMessageQueue* entnommen. Im Folgenden wird geprüft ob die Warteschlange nach Entnahme einer Nachricht wieder über freie Kapazitäten verfügt, die gegebenenfalls mit Elementen aus der Liste der wartenden *Sender* (*waitingSenders*) aufgefüllt werden kann. Sollte dies der Fall sein, wird ein *Sender* aus der Liste der Wartenden entnommen, seine Nachricht in die Warteschlange eingeführt und eine Benachrichtigung in Form einer *SendMessage* an den *Sender* verschickt.

```
1       internal protected T DequeueMessage<T>() where T : IMessage
2       {
3           lock (this)
4           {
5               if (this.internalMessageQueue.Count == 0)
6                   return default(T);
7               if (!(this.internalMessageQueue.Peek() is T))
8                   return default(T);
9               T msg = (T)this.internalMessageQueue.Dequeue();
10              while (this.internalMessageQueue.Count < this.MaxQueueLength)
11              {
12                  if (this.waitingSenders.Count == 0)
13                      break;
14                  Sender sender = this.waitingSenders[0];
15                  this.waitingSenders.RemoveAt(0);
16                  this.internalMessageQueue.Enqueue(sender.Message);
17                  sender.Protocol.BroadcastMessageReliably(new SendMessage() {
                        Sender = sender });
18              }
19              return msg;
20          }
21      }
```

Auflistung 3.4: Generische DequeueMessage Methode der Klasse AbstractMessageQueue

Die Klassen MessageQueue und BroadcastQueue

Das Gears4Net Framework stellt neben der abstrakten Basisklasse *AbstractMessageQueue* die beiden Varianten *MessageQueue* und *BroadcastQueue* bereit, die ein divergentes Nutzungsverhalten aufweisen. Die *BroadcastQueue* ist als Hauptnachrichtenschlange einer *Protocol*-Instanz, deren Funktionalität in Kapitel 3.6.5 aufgezeigt wird, konzipiert. Sie ist für das Zustellen (*broadcasting*) der eingehenden Nachrichten an alle *State-Machines* einer *Protocol*-Instanz zuständig. Sobald eine Nachricht an die *BroadcastQueue* übergeben wird, folgt der Aufruf der *Protocol.Wakeup*-Methode, die die Verarbeitung der Nachricht innerhalb der *Protocol*-Instanz anstößt. Sollte innerhalb der Instanz kein *Receiver* auf den Eingang der Nachricht registriert sein, so wird die Nachricht verworfen.

Das Verhalten der *MessageQueue* verläuft konträr. Die Klasse stellt Nachrichten ausschließlich an explizit über die *Receive*-Methoden der Klasse registrierte *Receiver* zu. Eine automatische Verteilung der Nachrichten über eine *Protocol*-Objekt, wie dieses bei der Implementierung der *BroadcastQueue* der Fall ist, ist bei der *MessageQueue* nicht vorgesehen.

3.6.2 Signals

Bei der Kommunikation zwischen zwei beliebigen Entitäten ist zu unterscheiden, ob diese dem Transfer einer Information oder lediglich der Anzeige, dass eine Information anliegt, dienen. Im ersten Fall enthält die übersandte Nachricht eine Information in Form eines Datums, das zu übertragen ist, im zweiten Fall ist die übertragene Nachricht bereits die Information selbst. Der zweite Fall kann somit auch als Signalisierung eines Ereignisses betrachtet werden.

Das Gears4Net Framework stellt die *Signal*-Klasse bereit, die als Signalgeber für verschiedenste Ereignisse genutzt werden kann. Ein *Signal* kann von einer Entität ausgelöst und von einer beliebigen, aber beschränkbaren Menge anderer Entitäten wahrgenommen werden. Dabei kann unterschieden werden, ob ein *Signal* nach seiner Auslösung direkt wieder in den alten Zustand überführt werden soll oder im ausgelösten Zustand verbleibt. Dieses Verhalten ist mit dem eines Leuchtturms vergleichbar. Dieser kann beispielsweise einem Schiff durch kurzzeitiges Einschalten einer Lampe eine Information signalisieren oder die Lampe dauerhaft anschalten, um auch anderen Schiffen zu einem späteren Zeitpunkt die Information bereit zustellen.

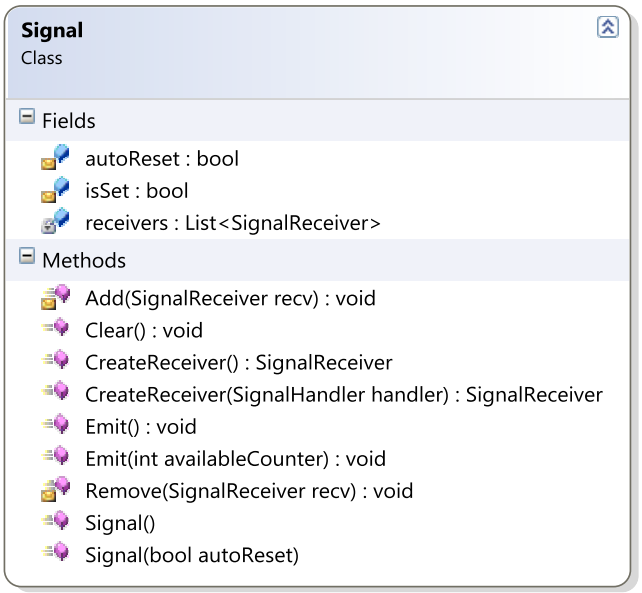

Abbildung 3.6: Klassendiagramm der Signal-Klasse

Die *Signal*-Klasse, deren Klassendiagramm in Abbildung 3.6 dargestellt ist, übernimmt die skizzierte Aufgabe. Der zweifach überladene Konstruktor der *Signal*-Klasse erlaubt die Angabe, ob ein *Signal* nach der Ausführung in den Ursprungszustand zurück überführt werden soll. Der Konstruktor stellt dazu den booleanschen

Parameter *autoReset* bereit, der vom Entwickler gesetzt werden kann. Im Standardfall ist die Eigenschaft mit dem Wert *wahr* initialisiert.

Um auf den Eintritt eines *Signals* zu warten, stellt die *Signal*-Klasse zwei Methoden bereit, die eine Instanz der in Kapitel 3.6.3 vorgestellten *SignalReceiver* erzeugen. Ein solcher *SignalReceiver* wird instantan benachrichtigt, sobald ein Ereignis ausgelöst wird. Die beiden *CreateReceiver*-Methoden mit unterschiedlicher Parameterliste erzeugen einen *Receiver* und geben die Instanz als Rückgabewert der Funktion zurück. Zudem erlaubt die *Signal*-Klasse das Hinzufügen von selbst erzeugten *SignalReceiver*-Objekten über den Aufruf der *Add*-Methode oder das Entfernen einer *Signal*-Wartebedingung durch Aufruf der *Remove*-Funktion.

Um ein *Signal* mittels eines *SignalReceivers* empfangen zu können, muss dieses von einer *Signal*-Instanz ausgelöst werden können. Die *Signal*-Klasse stellt dazu die *Emit*-Methode bereit. Die *Emit*-Funktion traversiert über eine Liste aller registrierten *SignalReceiver* und extrahiert die darin enthaltenen *Protocol*-Instanzen, denen das *Signal* zugestellt werden soll. Die Zustellung des *Signals* erfolgt über das Versenden einer *SignalMessage* an die beteiligten *Protocol*-Instanzen.

3.6.3 Receiver

Das Gears4Net Programmiermodell basiert auf dem Konzept einer Zustandsmaschine. Dieses sieht vor, dass eine Zustandsmaschine beziehungsweise ein Automat mittels einer Transition von einem Zustand a in einen Folgezustand b überführt werden kann, sobald eine valide Eingabe erfolgt.

Die Eingabe eines Automaten wird im Folgenden als Wartebedingung betrachtet. Die Maschine befindet sich solange in einem Zustand, bis die Wartebedingung für eine Eingabe erfüllt ist. Die Spezifikation einer solchen Wartebedingung erfolgt über *Receiver*. Diese können mittels Konjunktion und Disjunktion zu komplexen Wartebedingungen kombiniert werden.

Ein *Protocol* verarbeitet in jedem Schritt eine eingegangene Nachricht. Bei jedem Schritt werden alle aktiven *Receiver* der *State-Machines* eines *Protocols* über den Eingang dieser Nachricht informiert. Jeder *Receiver* entscheidet, ob die eingegangene Nachricht seine Wartebedinung erfüllt, seinen internen Zustand ändert und somit einen Teil der Wartebedingung erfüllt oder nicht von Relevanz ist. Im ersten Fall terminiert der *Receiver* und die ihn umhüllende *State-Machine* wird fortgesetzt. In den beiden anderen Fällen verbleibt der *Receiver* in der Liste der aktiven Wartebedingungen.

Typ	Kurzbeschreibung
ReceiverBase	Abstrakte Basisklasse für alle Receiver.
ReceiverBase <T>	Abstrakt generische Basisklasse für Nachrichten des Typs *IMessage*.
Receiver<T>	Receiver für Nachrichten des Types T mit optionaler Filterfunktionalität.
QueueReceiver <T>	Receiver für aufgestaute Nachrichten des Types T in einer MessageQueue
Sender	Wartet, bis eine Nachricht zugestellt wurde.
ProtocolReceiver	Wartet auf die Terminierung eines Protocols.
StateMachineReceiver	Wartet auf die Terminierung einer *State-Maschine*.
DetachedReceiver	Wartet auf die Terminierung einer als *detached* gestarteten *State-Maschine*.
SignalReceiver	Wartet, bis ein Signal emittiert wurde.
AndReceiver	Wartet auf die Terminierung aller übergebenen Receiver.
OrReceiver	Wartet auf die Terminierung eines der übergebenen Receiver.
KMultiReceiver	Wartet auf die Terminierung von k übergebenen Receivern.
SequentialReceiver	Verarbeitet einen Receiver, sobald sein Vorgänger in der Liste der übergebenen Receiver terminiert hat. Der Receiver ist auch als generisch typisierter SequentialReceiver <T> verfügbar.
PersistentReceiver	Ein sich automatisch neu startender Receiver.
ParallelReceiver	Nimmt eine definierte Menge an Nachrichten in beliebiger Reihenfolge an.
JoinReceiver	Wartet auf die Terminierung einer Menge von State-Machines.
IntervalReceiver	Wartet wiederkehrend auf den Ablauf eines definierten Zeitintervalls.
TimeoutReceiver	Wartet einmalig auf den Ablauf eines definierten Zeitintervalls.
AsyncResultReceiver	Verarbeitet das Ergebnis einer asynchron aufgerufenen Methode. Der Receiver ist auch als generisch typisierter AsyncResultReceiver <T> und AsyncResultReceiver <T, O> verfügbar.

Tabelle 3.4: Kurzbeschreibung der Receiver des Gears4Net Frameworks

Die Mächtigkeit der komplexen Wartebedingungen fußt auf der Funktionalität und Flexibilität verschiedenster *Receiver*-Typen. Das Gears4Net Framework stellt dazu 20 verschiedene *Receiver*-Typen in 5 Gruppen bereit, die in Tabelle 3.4 kurz vorgestellt werden. Die erste Gruppe beinhaltet die abstrakten Basistypen und bildet somit die oberste Hierarchieebene des *Receiver*-Konzeptes. In den Zuständigkeitsbereich der zweiten Gruppe fallen die *Receiver* mit einer atomaren Aufgabe, beispielsweise das Warten auf Nachrichten eines speziellen Typs. Die dritte Klassifikationsgruppe bilden die für die komplexen Wartebedingungen zuständigen *Receiver*. Diese ermöglichen unter anderem die angesprochenen Konjunktions- und Disjunktionsfähigkeiten. Die vorletzte Gruppe beinhaltet *Interval-* und *Timeout-Receiver*, die beispielsweise nach einem definierten Intervall auslösen. Abgeschlossen wird die Klassifikation mit der Gruppe der *Async-Receiver*, die die Verarbeitung von asynchron aufgerufenen Funktionen der Microsoft .NET Klassenbibliothek kapseln und vereinfachen.

3.6.3.1 Abstrakte Receiver Basisklassen

Die Grundlage der verschiedenen *Receiver* wird durch die Basisklassen *ReceiverBase* und *ReceiverBase<T>* gebildet, welche im Folgenden beschrieben werden.

ReceiverBase

Das gemeinsame Basiselement aller *Receiver*-Typen bildet die abstrakte Klasse *ReceiverBase*. Diese fasst die syntaktischen und funktionalen Gemeinsamkeiten aller *Receiver* in einer Klassenstruktur zusammen. Der Vorzug des Konzeptes der abstrakten Klasse gegenüber dem einer Schnittstelle als Basiselement ist durch die Tatsache begründet, dass eine abstrakte Klasse neben der Vorgabe der Signatur auch gemeinsam genutzte Funktionalität enthalten kann.

Das Klassendiagramm des abstrakten, nicht instanziierbaren Datentyps *ReceiverBase* ist in Abbildung 3.7 dargestellt. Die *Receiver*-Basisklasse implementiert drei elementare Funktionsblöcke „Initialisierung und Terminierung“, „Nachrichtenverarbeitung“ und „Modellierung komplexer Wartebedingungen“, die im Folgenden detailliert beschrieben werden.

Initialisierung und Terminierung

Der erste Funktionalitätsbaustein ist für die Initialisierung und spätere Terminierung der *Receiver* zuständig. Die *ReceiverBase*-Klasse stellt hierzu die öffentliche

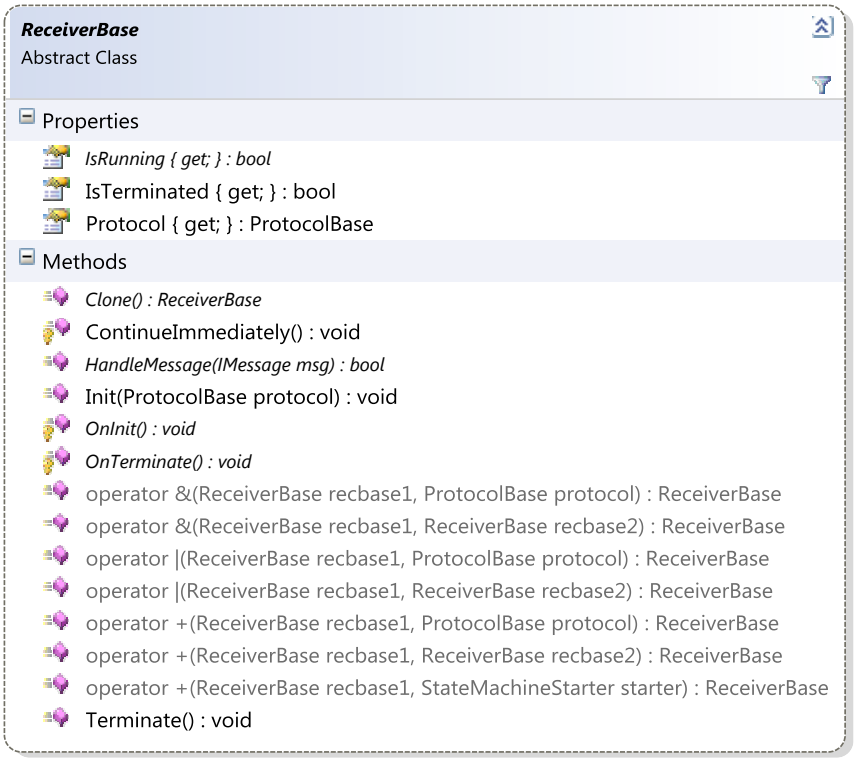

Abbildung 3.7: Klassendiagramm der ReceiverBase-Klasse

Methode *Init* und *Terminate* bereit und legt darüber hinaus fest, dass jeder *Receiver* die abstrakten Methoden *OnInit* und *OnTerminate* implementieren muss. Letztere werden automatisch bei der Initialisierung beziehungsweise der Terminierung eines *Receivers* aufgerufen und erlauben die Ausführung von *Receiver*-spezifischer Funktionalität. Im Zuge des Initialisierungsprozesses wird dem *Receiver* die Instanz des umschließenden *Protocols* übergeben, um diesem beispielsweise den Zugriff auf die *Message-Queue* des *Protocols* zu ermöglichen. Die Terminierung eines *Receivers* erfolgt durch den Aufruf der *Terminate*-Methode. Ein *Receiver* terminiert aus zweierlei Gründen: Erstens, falls die Wartebedingung für den *Receiver* erfüllt ist, und zweitens, insofern die Wartebedingung des *Receivers* nicht mehr benötigt wird. Unabhängig vom Grund der Terminierung ermöglicht der Aufruf der *OnTerminate*-Methode die Freigabe aller allokierten Ressourcen und somit die Durchführung einer effizienten Garbage-Collection und eine damit verbundene Vermeidung von Speicherlecks.

Nachrichtenverarbeitung

Die Verarbeitung eingehender Nachrichten fällt in den Zuständigkeitsbereich des zweiten Funktionsblocks. Die in der *Message-Queue* eines *Protocols* aufgestauten

Nachrichten werden sequentiell unter Berücksichtigung der Reihenfolge verarbeitet. In jedem Bearbeitungsschritt wird die aktuelle Nachricht an alle aktiven *Receiver* weitergegeben. Das umschließende Protocol ruft dazu die *HandleMessage*-Methode des jeweiligen *Receivers* auf. Die Methode von booleanschen Rückgabetyp erwartet die aktuelle Nachricht als Parameter. Sie evaluiert *wahr*, falls die eingegangene Nachricht die Wartebedingung des *Receivers* vollständig erfüllt, und *falsch*, insofern die Wartebedingung nicht erfüllt oder die Nachricht nicht für den *Receiver* bestimmt gewesen ist.

Der Lebenszyklus eines *Receivers* endet automatisch mit der Erfüllung seiner internen Wartebedingung. Der *Receiver* terminiert und gibt dabei jegliche allokierte Ressourcen frei und ermöglicht somit eine effektive *Garbage Collection*.

Die zweite für die Nachrichtenverarbeitung zuständige Methode *ContinueImmediatly* wird ausschließlich von den Methoden *OnInit* und *HandleMessage* aufgerufen. Die rein für den internen Gebrauch vorgesehene und deshalb als *protected* markierte Methode signalisiert dem umschließenden *Protocol*, dass dieses unverzüglich mit der weiteren Ausführung fortfahren kann, wie dies beispielsweise direkt nach der Initialisierung einer *State-Maschine* der Fall ist.

Modellierung komplexer Wartebedingungen

Die Forderung zur Modellierung komplexer und mächtiger Wartebedingungen in Form von exakten und zugleich intuitiven Ausdrücken wirkt zuerst konträr. Dieser Spagat wird dahingehend gelöst, dass dem Nutzer des Frameworks eine intuitive Operatorensyntax bereitgestellt wird, die intern durch einen Baum spezialisierter *Receiver* repräsentiert wird.

Das folgende Beispiel zeigt die Umwandlung einer komplexen Wartebedingung in Operatorensyntax in den intern verwalteten *Receiver*-Baum. Um das Verständnis der Baumstruktur zu ermöglich, ist es nötig, drei spezialisierte *Receiver* vorweg einzuführen. Eine tiefer gehende Betrachtung erfolgt im Anschluss. Zu diesen zählt der *Receiver*<*T*>, dessen Terminierungsbedingung durch den Eingang einer Nachricht eines definierten Typs T erfüllt ist, der *AndReceiver*, der terminiert, sobald der linke und rechte Operand terminieren, und der *OrReceiver*, dessen Terminierungsbedingung erfüllt ist, sobald einer seiner Operanden terminiert.

Die exemplarische Wartebedingung in diesem Beispiel erfordert den Eingang einer Nachricht der Typen A oder B und zudem den Eingang einer Nachricht vom Typ C. Die Variablen *a, b, c* repräsentieren jeweils einen *Receiver*<*T*>, der auf den Eingang

der Nachrichten vom Typ A, B oder C wartet. Die Operatorensyntax ermöglicht nun die kompakte Darstellung der Wartebedingung durch den Ausdruck *((a | b) & c)*.

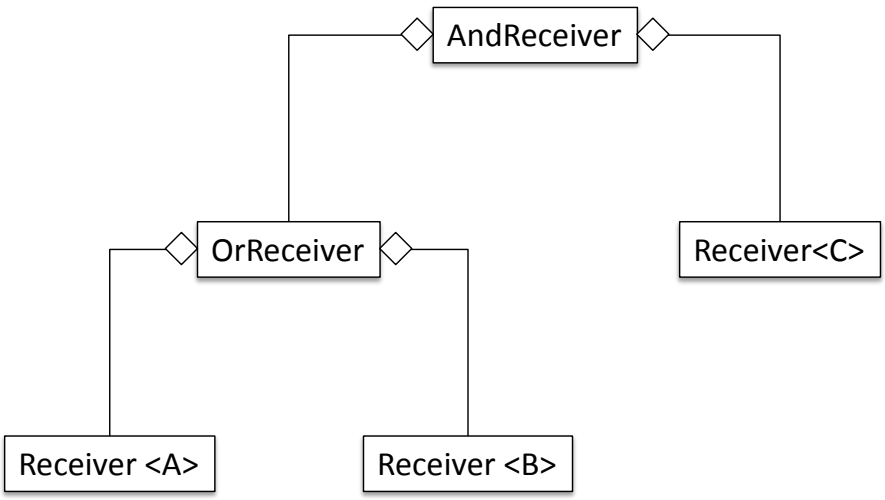

Abbildung 3.8: Darstellung der Wartebedingung *((a | b) & c)* als Receiver-Baumstruktur

Der Aufbau des in Abbildung 3.8 dargestellten *Receiver*-Baums lässt sich direkt aus dem oben genannten Ausdruck ableiten. Der additive *PLUS*-Operator bindet auf Grund der höchsten Präzedenz am stärksten und bildet somit die Wurzel des *Receiver*-Baums. Die beiden Operanden des *PLUS*-Operators *(a | b)* und *c* werden getrennt betrachtet. Der rechte Operator *c* kann als Blatt des Baumes direkt als *Receiver<C>* in den Baum eingehangen werden. Der linke Operand muss hingegen weiter in einen *OrReceiver* mit den Operanden *a* und *b* aufgespalten und in den Baum eingehangen werden. Bei den beiden Operanden des *OrReceiver* handelt es sich wiederum um Blätter des Baumes. Mit ihnen ist analog zum Operanden *c* zu verfahren. Das Verfahren terminiert, sobald kein weiterer Operator umzuwandeln beziehungsweise aufzusplitten ist.

Implementierung des Operatorenkonzeptes

Die *ReceiverBase*-Klasse bedient sich der in Kapitel 2.2 vorgestellten Technik des Überladens von Operatoren, um die im vorangegangenen Abschnitt beschriebene kompakte und effiziente Operatorensyntax zu ermöglichen. Zu diesem Zweck werden in der *ReceiverBase*-Klasse die beiden logischen Operatoren *UND* und *ODER* sowie der additive Operator *PLUS* mit den in Tabelle 3.5 angegebenen Parameterlisten überladen. Für die folgende Beschreibung der Implementierung der überschriebenen Operatoren wird jeweils ein Operator aus jeder der drei in Tabelle 3.5 dargestellten Gruppen herangezogen und exemplarisch betrachtet.

Operator	Receiver	Semantik
Parameterliste: *(ReceiverBase, ReceiverBase)*		
UND	AndReceiver	logisches UND
ODER	OrReceiver	logisches ODER
PLUS	SequentialReceiver	Konkatenation
Parameterliste: *(ReceiverBase, ProtocolBase)*		
UND	AndReceiver, ProtocolReceiver	logisches UND
ODER	OrReceiver, ProtocolReceiver	logisches ODER
PLUS	SequentialReceiver, ProtocolReceiver	Konkatenation
Parameterliste: *(ReceiverBase, StateMachineStarter)*		
PLUS	SequentialReceiver, StateMachineReceiver	Konkatenation

Tabelle 3.5: Abbildung der Operatoren auf spezialisierte Receiver

Die erste Gruppe ist durch zwei Operanden vom Type *ReceiverBase* charakterisiert. Der *ODER*-Operator wird beispielsweise auf den in Abschnitt 3.6.3.3 vorgestellten *OrReceiver* abgebildet. Ein *OrReceiver* erlaubt die Disjunktion von beliebig vielen Receivern. Der in Auflistung 3.5 dargestellte Quelltext prüft daher, ob es sich beim linken Operanden *(rec1)* bereits um einen *OrReceiver* handelt. Ist dies der Fall, so kann der rechte Operand als weiteres Element hinzugefügt werden. Andernfalls ist ein neuer *OrReceiver* zu erstellen, dem sowohl der rechte als auch der linke Operand angefügt werden.

Die Gruppe der Operatoren mit einem Parameter vom Typ *ReceiverBase* und einem zweiten vom Typ *ProtocolBase* erlauben es, auf die Terminierung eines ganzen *Protocols* oder eines *Receivers* zu warten. Der *UND*-Operator wird beispielsweise auf den *AndReceiver* abgebildet, der in Abschnitt 3.6.3.3 detailliert erläutert wird. Die Implementierung des Operators verläuft mit einer Ausnahme analog zu der ersten Gruppe. Der *AndReceiver* erlaubt ausschließlich die Konjunktion von *Receivern*. Da ein *Protocol* kein *Receiver* ist, muss ein auf die Terminierung von *Protocols* spezialisierter *Receiver* instanziiert und dem *AndReceiver* hinzugefügt werden. Diese Aufgabe wird durch den *ProtocolReceiver* (siehe Abschnitt 3.6.3.2) übernommen und ist in der Implementierungsillustration in Zeile 15 und 18 dargestellt.

Die dritte und letzte Gruppe ist ausschließlich durch den *PLUS*-Operator besetzt. Sie unterscheidet sich signifikant auf Grund des zweiten Parameters von den beiden vorherigen Gruppen. Bei dem als zweiten Parameter übergebenen Objekt des

```
1       public static ReceiverBase operator |(ReceiverBase rec1, ReceiverBase rec2)
2       {
3           if (rec1 is OrReceiver)
4           {
5               (rec1 as OrReceiver).Add(rec2);
6               return rec1;
7           }
8           return new OrReceiver(rec1, rec2);
9       }
10
11      public static ReceiverBase operator &(ReceiverBase rec1, ProtocolBase prot)
12      {
13          if (rec1 is AndReceiver)
14          {
15              (rec1 as AndReceiver).Add(new ProtocolReceiver(prot));
16              return rec1;
17          }
18          return new AndReceiver(rec1, new ProtocolReceiver(prot));
19      }
20
21      public static ReceiverBase operator +(ReceiverBase rec1,
            StateMachineStarter starter)
22      {
23          StateMachine sm = new StateMachine(starter);
24          ReceiverBase newRec = new StateMachineReceiver(sm);
25
26          if (rec1 is SequentialReceiver)
27          {
28              (rec1 as SequentialReceiver).Add(newRec);
29              return rec1;
30          }
31          return new SequentialReceiver(rec1, newRec);
32      }
```

Auflistung 3.5: Auszug der überladenen Operatoren der ReceiverBase-Klasse

Typs *StateMachineStarter* handelt es sich um einen typsicheren Funktionszeiger. Dieser zeigt auf eine Funktion, deren Syntax der in Kapitel 3.6.4 dargestellten *State-Machine* entspricht. Der *PLUS*-Operator konkateniert beide Operanden. Der dazu verwendete *SequentialReceiver* wartet solange, bis der jeweils erste *Receiver* der internen Liste terminiert, und stößt anschließend die Initialisierung des folgenden *Receivers* an. Da es sich im Falle des übergebenen *StateMachineStarter*-Objektes um den Zeiger auf eine *State-Machine* handelt, ist zuallererst eine solche zu erstellen und ein spezieller *StateMachineReceiver* (siehe Abschnitt 3.6.3.2) zu erzeugen, der auf die Terminierung dieser wartet. Der dazu notwendige Quelltext ist in den Zeilen 22 und 23 in Auflistung 3.5 dargestellt. Die folgende Implementierung des *SequentialReceivers* erfolgt analog zu den Implementierungen der Gruppen eins und zwei.

ReceiverBase<T>

Die abstrakte *ReceiverBase<T>*-Klasse bildet die zweite Indirektion in der Hierarchie der *Receiver*. Sie agiert analog zu ihrer Basisklasse *ReceiverBase* als Container für den gemeinsam genutzten Quellcode. Die verallgemeinerte Verarbeitung von Nachrichten, die die Schnittstelle *IMessage* implementieren, fällt in den Aufgabenbereich von *ReceiverBase<T>*. Der Typparamter T der Klasse unterliegt daher der Einschränkung, dass ausschließlich Typen für T eingesetzt werden dürfen, die *IMessage* implementieren.

Die generische *ReceiverBase<T>*-Klasse stellt zwei Funktionalitäten für ihre beiden Kinder *Receiver<T>* und *QueueReceiver<T>* bereit. Da jede von *ReceiverBase<T>* erbende Klasse für die Verarbeitung von Nachrichten des Typs *IMessage* zuständig ist, stellt diese die abstrakte Signatur des *Getter-Property ReceivedMessage* bereit, das von jeder erbenden Klasse implementiert werden muss.

Der zweite Funktionalitätsbaustein erweitert die Liste der überladenen *PLUS*-Operatoren mit Hilfe des in Auflistung 3.6 dargestellten generischen *SequentialReceiver<T>*. Das durch die beiden generisch typisierten Parameter *ReceiverBase<T>* und *StateMachineStarter<T>* und den in Absatz 3.6.3.3 erläuterten *SequentialReceiver<T>* gebildete Konstrukt ermöglicht die konkatenierte Ausführung von *Receiver* und *State-Machine*, wobei der Rückgabewert des *Receivers* als Parameter in die *State-Machine* eingeht.

```
1       public static ReceiverBase operator +(ReceiverBase<T> rec1,
            StateMachineStarter<T> starter)
2       {
3           return new SequentialReceiver<T>(rec1, starter);
4       }
```

Auflistung 3.6: Überladener PLUS-Operator der Klasse ReceiverBase<T>

Das Verwendung der generischen Klassen erlaubt die Prüfung auf Typsicherheit zur *Compile*-Zeit. Im gezeigten Beispiel kann somit sichergestellt werden, dass die vom *ReceiverBase<T>* erwartete Nachricht des Typs T der *State-Machine* mit einem identischen Typparameter übergeben werden kann. Das vorgestellte Verfahren verhindert mit der strengen Typisierung und der damit verbundenen Prüfung zur *Compile*-Zeit das Auftreten von Laufzeitfehlern.

3.6.3.2 Atomare Receiver

Die zweite Klassifikationsgruppe beinhaltet die atomaren *Receiver*. Diese sind auf das Auftreten eines singulären Ereignisses ausgelegt und terminieren im Anschluss. Im folgenden Abschnitt werden sieben atomare *Receiver* vorgestellt, die unter anderem auf den Eingang einer Nachricht eines speziellen Typs oder beispielsweise auf die Terminierung von Zustandsmaschinen oder Protokollen warten.

Receiver<T>

Der Nachrichten-*Receiver<T>* wartet auf den Eingang einer Nachricht des Typs *T*. Optional erlaubt der Receiver die Angabe eines Filterkriteriums sowie eines *Handlers*, der nach Erhalt einer Nachricht aufgerufen wird. Der *Receiver<T>* ist in der Klassenhierarchie unter der generischen *ReceiverBase<T>*-Klasse angesiedelt und erbt daher entlang der *Receiver*-Hierarchie.

Die Angabe des optionalen Filterkriteriums sowie des *Handlers* erfolgt über die beiden Parameter *filter* und *handler* der Konstruktoren. Die *Receiver<T>*-Klasse stellt dazu vier Konstruktoren bereit, die jegliche Kombination der beiden Parameter zulassen. Die beiden Parameter sind vom Typ *Filter<T>* beziehungsweise *ReceiveHandler<T>*. Bei beiden Typen handelt es sich um generische, typsichere Funktionszeiger, die in Auflistung 3.7 dargestellt sind und im Verlauf der Nachrichtenverarbeitung aufgerufen werden.

```
1       public delegate void ReceiveHandler<T>(T parameter);
2       public delegate bool Filter<T>(T parameter);
```

Auflistung 3.7: Signatur der Funktionszeiger ReceiveHandler und Filter

Die Verarbeitung der eingehenden Nachrichten erfolgt über die von der abstrakten Basisklasse *ReceiverBase* geerbte und in Auflistung 3.8 dargestellten Methode *HandleMessage*. Diese Methode mit booleanschem Rückgabetyp evaluiert *wahr*, sobald die Wartebedingung erfüllt ist, und andernfalls *falsch* (siehe Kapitel 3.6.3.1).

Innerhalb der *HandleMessage*-Methode wird deshalb zuallererst geprüft, ob der Typ der eingangenen Nachricht dem des generischen Typparamters *T* entspricht. Im Falle einer Übereinstimmung wird geprüft, ob die Nachricht den Kriterien des Filters genügt. Ist kein Filter angegeben, so gilt die Überprüfung automatisch als erfolgreich. Andernfalls wird, wie in Zeile 5 beschrieben, der Filter-Delegant aufgerufen. Der in der vorherigen Abbildung dargestellte Filter-Delegant zeigt auf eine Methode

mit booleanschem Rückgabewert. Diese evaluiert mit *wahr*, falls die übergebene Nachricht dem Filterkriterium entspricht.

Das Filterkriterium kann auf verschiedene Weisen implementiert werden. Diese reichen von Filtermethoden über anonyme Deleganten bis hin zu der sehr eleganten Variante des funktionalen Lambda-Ausdrucks, dessen Funktionsweise in [71] detailliert beschrieben wird.

Erfüllt eine Nachricht die geforderten Bedingungen bezüglich der Typ- und Filterkriterien, wird diese im *ReceivedMessage*-Property abgelegt und die *HandleMessage*-Methode evaluiert mit dem booleanschen Wert *wahr*. Sollte im Konstruktor des *Receivers* ein *Handler* übergeben worden sein, so wird dieser in Zeile 9 der Auflistung aufgerufen und ihm die aktuelle Nachricht übergeben. Für die übergebenen *Handler* gilt analog zu allen anderen Konstrukten innerhalb einer *Protocol*-Instanz, dass diese niemals blockieren dürfen.

```
1       public override bool HandleMessage(IMessage msg)
2       {
3           if (msg is T)
4           {
5               if (this.filter == null || this.filter((T)msg))
6               {
7                   this.ReceivedMessage = (T)msg;
8                   if (this.handler != null)
9                       this.handler(this.ReceivedMessage);
10                  return true;
11              }
12          }
13          return false;
14      }
```

Auflistung 3.8: Nachrichtenverarbeitung in der HandleMessage-Methode

QueueReceiver<T>

Die Verarbeitung von Nachrichten, die aus einer Warteschlange entnommen werden, obliegt dem *QueueReceiver*<*T*>. Dieser wartet solange, bis die in Kapitel 3.6.1 beschriebene *DequeueMessage*<*T*>-Methode der *MessageQueue*-Klasse eine Nachricht aus der Warteschlange entnimmt, die dem angeforderten Typ T entspricht. Sobald eine solche Nachricht erkannt wurde, wird der *QueueReceiver*<*T*> aktiv, ruft einen *Handler* auf und terminiert.

In der Klassenhierarchie steht der *QueueReceiver*<*T*> direkt unter der *ReceiverBase*<*T*> und implementiert das durch die Vererbungshierarchie vor-

gegebene Eigenschaftsfeld *ReceivedMessage*. Der Konstruktor der Klasse erlaubt die Übergabe eines Funktionszeigers, der bei Eingang einer Nachricht aufgerufen wird, und des *MessageQueue*-Objekts, auf dem der *Receiver* ausgeführt werden soll. Die Registrierung des *Receivers* bei der *Message-Queue* erfolgt in der *OnInit*-Methode der *QueueReceiver<T>*-Klasse. Das Abmelden erfolgt analog in der *OnTerminate*-Methode derselben Klasse. Die Terminierung der *QueueReceiver<T>*-Instanz erfolgt entweder explizit durch den Aufruf der *OnTerminate*-Methode oder implizit, da der *Reiceiver* automatisch terminiert wird, sobald keine Instanz auf das Resultat des *Reiceivers* wartet.

Sender-Receiver

Die asynchrone Nachrichtenverarbeitung des Gears4Net Frameworks erfolgt über die in Kapitel 3.6.1 vorgestellten Warteschlangen. Das Versenden einer Nachricht an ein anderes oder das eigene Protokoll ist als das Hinzufügen eines Objektes an eine Warteschlange zu betrachten. Eine Nachricht kann solange einer *MessageQueue* hinzugefügt werden, bis die Warteschlange an ihre Kapazitätsgrenze stößt. Ist die maximale Kapazität erreicht, kann ein *Sender-Receiver* eingesetzt werden. Dieser wartet solange, bis die Warteschlange wieder über ausreichend Kapazität verfügt, um das Element anzufügen. Der *Sender* terminiert, wenn die Nachricht in die Warteschlange aufgenommen wurde.

Die *Sender*-Klasse stellt zwei überladene Konstruktoren bereit, die die Registrierung des *Receivers* sowohl auf einem Protokoll als auch auf einer Warteschlange ermöglichen. Das erfolgreiche Versenden einer Nachricht kann zudem über das Eigenschaftsfeld *HasMessageBeenSent* erfragt werden.

ProtocolReceiver

Die Terminierung eines Protokolls wird durch den *ProtocolReceiver* überwacht. Diesem wird die zu überwachende Instanz des *Protocols* im Konstruktor übergeben und in einem Feld gespeichert. Die Terminierung eines Protokolls kann zweierlei Gründe haben. Erstens kann die initiale Zustandsmaschine des Protokolls in ihren Endzustand überführt, und zweitens kann das Protokoll aufgrund eines externen Eingriffs zur Terminierung gezwungen worden sein. Bei der Verwendung des *ProtocolReceivers* ist anzumerken, dass der Aufruf der *OnTerminate*-Methode zur Auslösung einer *TerminateMessage* führt. Das *Gears4Net*-Modell verwendet diesen Nachrichtentyp, um alle Untermaschinen und deren *Receiver* einer *Protocol*-Instanz terminieren zu lassen. Die *Receiver* terminieren nach der Verarbeitung der *TerminateMessage* instantan und verarbeiten keine weiteren Nachrichten.

StateMachineReceiver

Die Terminierungskontrolle von Zustandsmaschinen wird vom *StateMachineReceiver* übernommen. Dieser *Receiver*-Typ registriert sich auf der im Konstruktor übergebenen *State-Machine* und wartet solange, bis die Zustandsmaschine ihren Endzustand erreicht hat. Der *Receiver* prüft dazu in jedem Nachrichtenverarbeitungsschritt, ob die *State-Machine* in den finalen Endzustand übergegangen ist. Die Terminierung einer *State-Machine* kann zudem über den Aufruf der *OnTerminate*-Methode des *StateMachineReceiver* erzwungen werden. Sollte sich der Automat beim Aufruf der*OnTerminate*-Methode bereits in einem Endzustand befinden, kann die *OnTerminate*-Methode an dieser Stelle abgebrochen werden. Andernfalls wird mit der Zustellung der *TerminateMessage* begonnen, die den Automaten zur instantanen Terminierung zwingt.

DetachedReceiver

Das Gears4Net Konzept erlaubt die Ausführung von parallelen Automaten, die über das Kommando *LaunchDetached* innerhalb eines Protokolls erzeugt werden können. Die Überprüfung, ob eine solche Zustandsmaschine terminiert ist, erfolgt über den *DetachedReceiver*. Die Verwendung des *DetachedReceivers* erfolgt analog zu der des *StateMachineReceivers* mit der Ausnahme, dass im Fall des Aufrufs der *OnTerminate*-Methode keine Terminierungsnachricht an den Automaten übermittelt wird, sondern dieser in die Liste der *DetachedStateMachines* eines Protokolls aufgenommen und dort weiter verwaltet wird. Dieses Verhalten ist notwendig, da ein *Protocol* ansonsten keine Referenz auf eine mit *LaunchDetached* gestartete *State-Machine* besitzen und der Automat unter Umständen vom *Garbage-Collector* entsorgt werden würde.

SignalReceiver

Besondere Ereignisse, beispielsweise die Anforderung der Zustandsmaschinen- oder Protokollterminierung, können durch *Signals* (siehe Kapitel Kapitel 3.6.2) angezeigt werden. Der Aufruf der *Emit*-Funktion eines Signals führt zur Auslösung einer *SignalMessage*, die durch einen *SignalReceiver* empfangen und verarbeitet wird. Dem *SignalReceiver* werden dazu das zu überwachende Signal sowie ein Funktionszeiger übergeben. Die Registrierung eines solchen *Receivers* bei einem entsprechenden Signal erfolgt innerhalb der *OnInit*-Methode. Der Aufruf der *OnTerminate*-Methode führt analog zu einer Freigabe des Signals.

Die *Signal*-Klasse verfügt wie in Kapitel 3.6.2 beschrieben über eine *CreateReceiver*-Methode. Diese erzeugt beim Aufruf automatisch einen *SignalReceiver*, der direkt

auf das Signal-Objekt zeigt, auf dem es aufgerufen wurde, und ermöglicht somit den möglichst kompakten Umgang mit Signalen.

3.6.3.3 Komplexe Receiver

Die Gruppe der komplexen beziehungsweise zusammengesetzten *Receiver* bildet die Grundlage für die Verkettung von *Receivern*. Die einzelnen Elemente dieser Gruppe werden innerhalb des Gears4Net Frameworks verwendet, um das Konzept der Operatorensyntax zu ermöglichen. Die im Folgenden vorgestellten acht *Receiver* werden zumeist durch die überladenen Operatoren implizit erzeugt. Es jedoch auch möglich, diese *Receiver* manuell mit dem Schlüsselwort *new* zu instantiieren.

AndReceiver

Die Konjunktion von verschiedenen *Receivern* beliebigen Typs wird durch den *AndReceiver* ermöglicht, welcher zudem die Grundlage für den in Kapitel 3.6.3.1 vorgestellten *UND*-Operator bildet. Der *AndReceiver* ist jedoch im Gegensatz zum *UND*-Operator nicht auf zwei Operanden beschränkt und ermöglicht daher die Konjunktion von beliebig vielen *Receivern*.

Der *AndReceiver* verwaltet alle zu konjugierenden *Receiver* in einer Liste und terminiert genau dann, wenn alle *Receiver* der Liste terminiert sind. Die Klasse des *AndReceivers* stellt dazu zwei überladene Konstruktoren bereit. Der erste bildet den Spezialfall des *UND*-Operators mit zwei Operanden ab, der zweite die Generalisierung mit einer beliebigen Anzahl an *Receivern*.

An einen bestehenden *AndReceiver* können durch den Aufruf der *Add*-Methode weitere *Receiver* hinzugefügt werden. Diese Eigenschaft wird beispielsweise bei der Implementierung des in Auflistung 3.5 dargestellten *UND*-Operators ausgenutzt, um bei vielen Konjunktionen lediglich ein *AndReceiver*-Objekt verwenden zu müssen. Auf die Liste der *Receiver* eines *AndReceivers* kann über das Eigenschaftsfeld *Receivers* zugegriffen werden.

OrReceiver

Der strukturelle Aufbau des *OrReceivers* folgt dem des *AndReceivers* mit dem Unterschied, dass der *Receiver* die Disjunktion und nicht die Konjunktion implementiert. Der *OrReceiver* terminiert also genau dann, wenn einer der verwalteten *Receiver* terminiert.

Die Verwendung des *OrReceiver* erfolgt ebenfalls analog zur Verwendung des *AndReceivers*. Ein Ausdruck mit drei *Receivern* a, b, c der Form $a \mid b \mid c$ kann wie in Auflistung 3.9 aufgezeigt auf drei unterschiedlichen Wegen in einen *OrReceiver* überführt werden. Die Zeile eins zeigt die durch die Operatorensyntax gegebene Schreibweise, die direkt dem oben angegebenen Ausdruck folgt. Die Zeilen 3 und 4 zeigen das Standardverhalten, das bei der Implementierung des logischen *OR*-Operators zugrundegelegt wird. Für die erste Disjunktion wird dabei ein *OrReceiver* erstellt. Bei jeder weiteren direkt folgenden Disjunktion wird der Operand direkt zu dem bestehenden *OrReceiver* hinzugefügt. Dieses Verhalten optimiert sowohl den Speicherverbrauch als auch die Geschwindigkeit der Ausdrucksverarbeitung. Die zweite Nutzungsmöglichkeit des *OrReceivers* wird in Zeile 6 des Beispiels dargelegt und ist besonders für den direkten Gebrauch des *Receivers* vorgesehen.

```
1   ReceiverBase rec1 = a | b | c;
2
3   ReceiverBase rec2 = new OrReceiver(a, b);
4   rec2.Add(c);
5
6   ReceiverBase rec3 = new OrReceiver(new ReceiverBase[] {a, b, c});
```

Auflistung 3.9: Methoden der Verwendung des OrReceivers

KMultReceiver

Die Generalisierung des *OrReceivers* wird durch den *KMultReceiver* erreicht. Dieser terminiert im Gegensatz zum *OrReceiver* nicht, wenn lediglich ein *Receiver* terminiert, sondern wenn eine definierte Anzahl k aus einer Menge von n *Receivern* terminiert. Ein *KMultReceiver* mit *k=1* würde somit einem *OrReceiver* entsprechen.

Der strukturelle Aufbau des *KMultReceiver* ist an die beiden vorangegangenen *Receiver* angelehnt. Er propagiert jedoch die bereits terminierten *Receiver* im Eigenschaftsfeld *CompletedReceivers* und erlaubt somit den Zugriff auf die Daten der schon zu Ende gelaufenen Wartebedingungen.

SequentialReceiver

Das im Kapitel 3.6.3.1 vorgestellte Konzept der allgemeinen *Receiver*-Konkatenation mittels des überladenen *PLUS*-Operators fußt auf dem *SequentialReceiver*. Ein *SequentialReceiver* verwaltet eine *Receiver*-Liste und arbeitet diese Element für Element ab. Der *SequentialReceiver* initialisiert den ersten *Receiver* einer Liste und stellt diesem solange Nachrichten zu, bis dieser terminiert. Nach jeder erfolgreichen

Terminierung eines *Receivers* initialisiert der *SequentialReceiver* das jeweils nächste *Receiver*-Objekt der Liste. Das Verfahren wird solange fortgesetzt, bis der letzte *Receiver* terminiert ist.

SequentialReceiver<T>

Die Entwicklung der generischen Variante des *SequentialReceivers* unterscheidet sich signifikant von der allgemeinen Version, die im vorherigen Paragraph beschrieben wurde. Ziel des generischen *SequentialReceiver*<*T*> ist es, nach Eingang einer Nachricht einen neuen Unter- beziehungsweise Teilautomaten zu erzeugen und die eingegangene Nachricht als Parameter an den Teilautomaten zu übergeben. Die Implementierung *SequentialReceiver*<*T*> bildet somit die Grundlage für die Semantik des überladenen *PLUS*-Operators der generischen *ReceiverBase*<*T*>-Klasse.

Der *SequentialReceiver*<*T*> erlaubt die Konkatenation der generischen Typen *ReceiverBase*<*T*> und *StateMachineStarter*<*T*> genau dann, wenn beide Typen einen identischen Typ für T einsetzen. Die Identitätsprüfung des Typparameters T wird sowohl bei der direkten als auch bei der indirekten Verwendung des *SequentialReceiver*<*T*> mittels des überschriebenen *PLUS*-Operator automatisiert vom C#-Compiler übernommen. Im Falle einer fehlenden Übereinstimmung der Typparameter führt dieses automatisch zu einem Compilerfehler während des Übersetzungsverfahrens.

Im Konstruktor des *SequentialReceivers*<*T*> können die beiden von der abstrakten Basisklasse *ReceiverBase*<*T*> erbenden Typen *Receiver*<*T*> und *QueueReceiver*<*T*> eingesetzt und mit einem *IMessage* implementierenden Datentypen typisiert eingesetzt werden. Der zweite Parameter des Konstruktors erwartet einen Funktionszeiger des in Auflistung 3.10 dargestellten Typs *StateMachineStarter*<*T*>. Anhand der Signatur des Deleganten ist erkennbar, dass es sich bei diesem um einen Zeiger auf einen der im Kapitel 2.6 vorgestellten Iteratoren mit dem Rückgabetyp *ReceiverBase* handelt. Die Parameterliste des Funktionszeigers erwartet die zwei Parameter der Typen *AbstractStateMachine* und T, wobei der Typ T dem generischen Typisierungsparameter der Methode entspricht.

```
1       public delegate IEnumerator<ReceiverBase> StateMachineStarter<T>(
            AbstractStateMachine machine, T parameter);
```

Auflistung 3.10: Generischer Funktionszeiger StateMachineStarter

Mit dem Zeitpunkt der Initialisierung des *SequentialReceivers<T>* verarbeitet dieser eingehende Nachrichten in der *HandleMessage*-Methode. Der *SequentialReceiver<T>* reicht eingehende Nachrichten solange an den im ersten Parameter im Konstruktor übergebenen *Receiver* des Typ *ReceiverBase<T>* weiter, bis dieser terminiert, und speichert die für die Terminierung relevante Nachricht. Die sequentielle Verarbeitung wird durch die Erzeugung einer neuen Zustandsmaschine fortgeführt. Der *SequentialReceiver<T>* erzeugt dazu eine *AbstractStateMachine<T>* und übergibt dieser den im Konstruktor angegebenen Funktionszeiger sowie die gespeicherte Nachricht als Parameter.

Das beschriebene Verfahren erlaubt es nun, das Ergebnis des ersten *Receivers* eines Operatorenausdrucks der Form *ReceiverBase<T> + StateMachinePointer* als Parameter der Funktion zu übergeben, auf die der *StateMachinePointer* zeigt. Der *SequentialReceiver<T>* terminiert mit Beendigung der erzeugten Zustandsmaschine. Daher wird in der *HandleMessage*-Methode ein *StateMachineReceiver* erzeugt, der auf die Terminierung der Maschine wartet und anschließend den *SequentialReceiver<T>* selbst terminiert.

PersistentReceiver

Die bisher vorgestellten *Receiver* reagieren auf das Eintreffen eines singulären Ereignisses. Die Implementierung einer Wartebedingung, die auf alle jemals eingehenden Ereignisse eines Typs wartet, ist mit den bisherigen *Receivern* lediglich mit einer Schleife realisierbar. Der *PersistentReceiver* schließt genau diese Lücke. Dem *Receiver* wird dazu im Konstruktor ein beliebiger, von *ReceiverBase* abgeleiteter *Receiver* übergeben und im privaten Eigenschaftsfeld *receiver* abgelegt.

Die Verarbeitung der eingehenden Nachrichten erfolgt anhand der in Auflistung 3.11 dargestellten *HandleMessage*-Methode. Die an den *PersistentReceiver* adressierte Nachricht wird dabei an den intern gespeicherten *Receiver* weitergegeben und von diesem verarbeitet. Dieses Verfahren wird solange durchgeführt, bis der interne *Receiver* terminiert. Die Aufgabe des *PersistentReceivers* besteht darin, den terminierten *Receiver* neu zu starten und somit eine identische Wartebedingung zu erzeugen. Dieses wiederkehrende Verfahren implementiert eine permanent gültige Wartebedingung. Der Prozess der Reinitialisierung eines *Receivers* durch den niemals terminierenden *PersistentReceiver* wird in den Zeilen 8 und 9 der Auflistung 3.11 dargestellt.

```
1       public override bool HandleMessage(IMessage msg)
2       {
3           if (this.IsTerminated)
4               return true;
5
6           if (this.receiver != null)
7           {
8               if (this.receiver.HandleMessage(msg))
9                   this.receiver.Init(this.Protocol);
10          }
11          return false;
12      }
```

Auflistung 3.11: HandleMessage-Methode des PersistentReceivers

ParallelReceiver

Bei der Entwicklung verteilter Systeme wird häufig die parallele Verarbeitung einer definierten Nachrichtenmenge gefordert. Das Beispiel eines Web-Servers verdeutlicht dieses. Ein Server verarbeitet eine Menge an parallelen *HTTP-GET*-Anfragen. Um die maximale Auslastung des Servers nicht zu überschreiten ist es notwendig, die Höchstgrenze der simultanen Verarbeitung zu beschränken.

Der *ParallelReceiver* kommt dieser Forderung nach. Die Konfiguration des *Receivers* ermöglicht die Angabe der maximal zu verarbeitenden Nachrichten. So ist beispielsweise zu spezifizieren, dass ein *Receiver* auf 100 Nachrichten des Typs *A* wartet, aber nur *5* Nachrichten parallel verarbeitet und zudem die Verarbeitung abbricht, sobald das Signal *softTerm* emittiert. Die Wartebedingungen die innerhalb des *ParallelReceiver* angegeben werden, können auch von komplexer Natur sein. Das in Auflistung 3.12 beschriebene Szenario erweitert das gegebene Beispiel um die Ausführung eines Subautomatens. Die dargestellte Wartebedingung ist erst dann erfüllt, wenn alle 100 Unterautomaten terminiert haben.

```
1       public IEnumerator<ReceiverBase> Execute(AbstractStateMachine sM)
2       {
3           yield return new ParallelReceiver(100, 5, new Receiver<A>() +
                myStateMachine, softTerm.CreateReceiver());
4       }
```

Auflistung 3.12: Verwendung eines ParallelReceivers

Das Klassendiagramm des *ParallelReceiver* in Abbildung 3.9 zeigt vier unterschiedliche Konstruktoren, die die problemspezifische Konfiguration des *Receivers* ermöglichen. Die Konfiguration erfolgt über die vier Parameter *parallelCount*, *complete-*

Count, *job* und *softTerm*, die wahlweise im Konstruktor angegeben oder mit Standardwerten initialisiert werden. Der Parameter *completeCount* gibt die Anzahl der maximal zu verarbeitenden Nachrichten an. Soll die Anzahl der Nachrichten nicht beschränkt werden, so kann der Parameter mit der Konstante *Int32.MaxValue* initialisiert werden. Der zweite Parameter, *parallelCount*, gibt die Anzahl der parallel zu verarbeitenden Nachrichten an. Wird ein Konstruktor gewählt, bei dem dieser Parameter nicht gesetzt wird, so werden automatisch beliebig viele Nachrichten parallel abgearbeitet. Der dritte Parameter spezifiziert die Wartebedingung, die für den *ParallelReceiver* gültig ist. Mit dem letzten Parameter kann eine weiche Terminierungsbedingung festgesetzt werden. Dieses Konstrukt erlaubt es festzulegen, dass ab der Erfüllung der weichen Terminierungsbedingung keine neuen *Receiver* instantiiert werden. Die bisher erzeugten parallelen *Receiver* werden allerdings bis zu ihrer eigenen Terminierung ausgeführt.

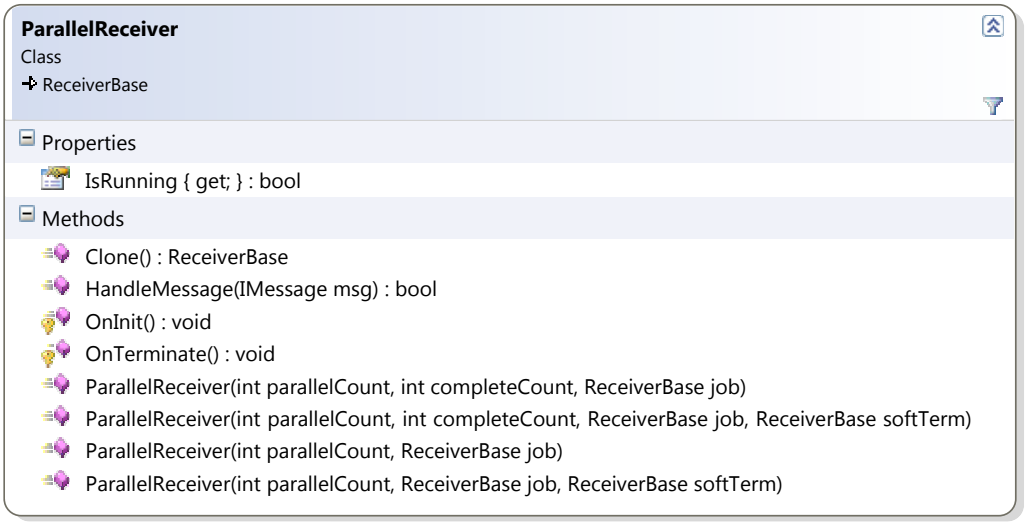

Abbildung 3.9: Klassendiagramm der ParallelReceiver-Klasse

Die Implementierung der Nachrichtenverarbeitung des *ParallelReceivers* gehört zu den komplexesten der vorgestellten *Receivern*. Innerhalb der *HandleMessage*-Methode des *ParallelReceivers* wird auf den Eingang einer Nachricht gewartet, die die Wartebedingung des im Konstruktor übergebenen *Jobs* erfüllt. Sobald der *Receiver* eine erste Nachricht konsumiert hat, wird der *Job* zur Liste der aktuell ausgeführten *Receiver* hinzugefügt. Sollte keine der drei Limitierungsfakten *parallelCount*, *completeCount* oder *softTerm* gegen die Registrierung eines neuen *Receivers* sprechen, wird der *Receiver-Job* geklont. Mit dem neuen *Job*-Objekt wird somit eine identische Wartebedingung erzeugt und das Verarbeitungsverfahren beginnt von vorn. Bei jeder Zustellung einer Nachricht in der *HandleMessage*-Methode wird die Liste der aktiven *Receiver* durchlaufen. Sollte diese mittlerweile terminierte *Recei-*

ver enthalten, so werden diese aus der Liste entfernt, um Platz für weitere parallel auszuführende *Receiver* zu schaffen.

JoinReceiver

Der *JoinReceiver* ist für den Umgang mit Zustandsmaschinen konzipiert. Der Lebenszyklus dieses *Receivers* ist darauf ausgelegt auf die Terminierung einer beliebigen Menge von *AbstractStateMachines* zu warten. Diese Bedingung ist gleichbedeutend mit dem erreichen der Endzustände aller Maschinen.

Die Implementierung des *Receivers* erwartet im Konstruktor eine *Array* von *AbstractStateMachines*. Während des Initialisierungsprozesses des *Receivers* wird dieses Feld durchlaufen und es wird für jede aktive Zustandsmaschine ein *TerminatedEvent*-Handler registriert. Die zugehörige Methode wird bei der Terminierung einer jeden Maschine aufgerufen. Sobald alle Automaten terminiert sind, wird eine *JoinMessage* erzeugt und in die Protokollwarteschlange eingefügt. Diese leitet die Terminierung des *JoinReceivers* ein, da mit dem Nachrichteneingang dessen Wartebedingung erfüllt ist.

3.6.3.4 Zeit und Intervall Receiver

Die beiden zeitgebenden *Receiver TimeoutReceiver* und *IntervalReceiver* ermöglichen den einfachen und zugleich sehr effektiven Umgang mit Zeitgebern in einem Gears4Net Protokoll. Beide *Receiver* sind dabei auf die vom Betriebssystem beziehungsweise von der Plattform des Microsoft .NET Frameworks vorgegebenen Zeitgeber angewiesen.

Das Microsoft .NET Framework stellt die drei in Kapitel 2.4 vorgestellten Zeitgeber *System.Windows.Forms.Timer*, *System.Timers.Timer* und *System.Threading.Timer* bereit, aus denen einer für die Implementierung der zeitgebenden *Receiver* ausgewählt werden muss. Der erste Zeitgeber ist speziell für die Verwendung in graphischen Steuerelementen konzipiert und kommt daher für die Verwendung innerhalb eines *Receivers* nicht in Betracht. Der zweite Zeitgeber basiert auf dem zuletzt genannten *System.Threading.Timer* und erweitert diesen lediglich um ein Synchronisierungsobjekt, dem im vorliegenden Fall keine Beachtung geschenkt werden muss. Die Wahl des Zeitgeberkonzeptes fällt auf den *System.Threading.Timer*, der vom Microsoft .NET Framework über die Windows API direkt auf *Hardware-Timer* abgebildet wird und somit die maximale Genauigkeit der vorgestellten Zeitgeber bietet.

Die direkte Abbildung des Zeitgebers auf Hardware Ressourcen obliegt jedoch der Beschränkung, dass nur eine endliche Zahl an *Hardware-Timern* simultan angefordert werden können. Werden mehr Zeitgeber angefordert als *Hardware-Timer* bereitstehen, so blockiert die aufrufende Funktion solange, bis eine Timer-Ressource freigeworden ist. Da das Konzept des *Gears4Net*-Frameworks keinerlei blockierende Aufrufe zulässt, ist die Implementierung einer logischen *Timer*-Klasse [118] unumgänglich. Das *Gears4Net*-Framework stellt dazu die Klasse *HighPerformanceTimer* bereit, die jedes angeforderte Zeitintervall in einer internen, geordneten Warteschlange verwaltet. Bei der Abarbeitung wird jeweils die Differenz zwischen dem letzten, der Schlange entnommenen Zeitintervall und dessen Nachfolger berechnet. Diese wird im Anschluss auf einen einzige *Hardware-Timer* abbildet.

TimeoutReceiver

Ein *TimeoutReceiver* verfügt intern über einen *HighPerformanceTimer*, der nach einer definierten Zeit auslöst und den *TimeoutReceiver* terminieren lässt. Der *TimeoutReceiver* verfügt über einen öffentlichen Konstruktor, in dem das Zeitintervall bis zur Terminierung und der im Anschluss an die Terminierung aufzurufende *Event-Handler* übergeben werden. Die Lebenszeit des *TimeoutReceivers* beginnt mit Aufruf der *OnInit* abzulaufen. Sobald die *Callback*-Methode *OnTimeout* durch den *HighPerformanceTimer* aufgerufen wird, erzeugt der *Receiver* eine *TimeoutMessage* und fügt diese der Protokollnachrichtenschleife hinzu. Durch die Zustellung der *TimeoutMessage* über das Warteschleifenkonstrukt des Protokolls wird dem Umstand Rechnung getragen, dass die Rückrufmethode des *HighPerformanceTimers* im Kontext des *Timer*-Threads aufgerufen wird und dieses konträr zu der Forderung steht, dass nur jeweils ein Thread in einem Protokoll aktiv sein darf.

Der *TimeoutReceiver* wartet im Anschluss an die Einstellung der Nachricht in die Warteschlange auf die Verarbeitung derselben. Sobald die *TimeoutMessage* den *Receiver* erreicht, ruft dieser den übergebenen *Handler* auf und terminiert. Die Verwendung des *TimeoutReceivers* erfolgt im Allgemeinen im Zusammenspiel mit anderen *Receivern*. Es ist beispielsweise üblich, auf den Eingang einer Nachricht des Typs *A* oder das Ablaufen eines Zeitintervalls zu warten. Unter Verwendung der vorgestellten überladenen Operatoren erlaubt dies kurze und präzise Ausdrücke der Form *Receive<A> | Timeout(1000)*.

IntervalReceiver

Die Aufgabe und Funktionalität des *IntervalReceivers* ähnelt der des *TimeoutReceivers* mit dem Unterschied, dass der *IntervalReceiver* immer wiederkehrende Ereignis-

se auslöst und niemals von selbst terminiert. Die Konfiguration des *IntervalReceivers* unterscheidet zwei Zeitintervalle. Das erste gibt die Zeit an, bis der *Receiver* zum ersten mal auslöst, das zweite die Zeit, die zwischen den wiederkehrenden Ereignissen vergeht.

Die Implementierung des *IntervalReceivers* stellt zwei unterschiedliche Konstruktoren bereit, die sich ausschließlich im jeweils letzten Parameter unterscheiden. Die ersten beiden Parameter sind für das initiale beziehungsweise wiederkehrende Zeitintervall vorgesehen. Der dritte Parameter des ersten Konstruktors ermöglicht die Übergabe eines Funktionszeigers, der nach Ablauf eines Intervalls aufgerufen werden soll. Der zweite Konstruktor hingegen erfordert einen Zeiger des Typs *StateMachineStarter*, der zum Start eines neuen Teilautomaten verwendet wird.

Die Verwendung der beiden unterschiedlichen Nutzungskonzepte wird in Auflistung 3.13 dargestellt. Der erste der beiden *IntervalReceiver* startet mit einer Verzögerung von 1000 Millisekunden und löst anschließend in einem Intervall von 2 Sekunden aus. Bei jeder Auslösung wird die Methode *IntervallCallback* aufgerufen und der darin enthaltene Quellcode ausgeführt. Der zweite *IntervallReceiver* startet ohne initiale Verzögerung und ruft im Anschluss die Zustandsautomaten *SubStateMachine* in einem Intervall von 4 Sekunden auf. Sollte die Ausführung einer erzeugten *SubStateMachine* länger als das spezifizierte Intervall andauern, wird solange kein Intervall ausgelöst, bis die *SubStateMachine* terminiert ist.

```
1   public IEnumerator<ReceiverBase> Execute(AbstractStateMachine sM)
2   {
3       yield return new IntervalReceiver(1000, 2000, IntervallCallbackPointer)
            | new IntervalReceiver(0, 4000, SubStateMachine);
4   }
5
6   public void IntervallCallback()
7   {
8       // Add custom code
9   }
10
11  public IEnumerator<ReceiverBase> SubStateMachine(AbstractStateMachine sM)
12  {
13      // Add custom code
14      yield break;
15  }
```

Auflistung 3.13: Beispielhafte Nutzung des IntervalReceivers

3.6.3.5 AsyncResult Receiver

Das Konzept des Gears4Net Frameworks fordert die ausschließliche Verwendung von nicht blockierenden Funktions- und Methodenaufrufen. Diese Forderung ist gleichbedeutend mit der Nutzung der asynchronen API [18] des Microsoft .NET Frameworks, die aufgrund der verwendeten Architektur zu einer Vielzahl an *Callback*-Methoden führt. Der Aufruf einer asynchronen Funktion besteht aus drei Bestandteilen: der *Begin...* und der *End...*-Methode sowie einem Funktionszeiger auf eine Rückrufmethode, die im Sequenzdiagramm in Abbildung 3.10 abgebildet sind. Die Parameterliste der *Begin...*-Methode beinhaltet neben den für die Methode relevanten Parametern auch den Zeiger auf die Methode, die aufgerufen werden soll, sobald das asynchrone Ereignis eintritt. Anschließend wird die *End...*-Methode aufgerufen, die das Ergebnis des asynchronen Aufrufs zurückliefert.

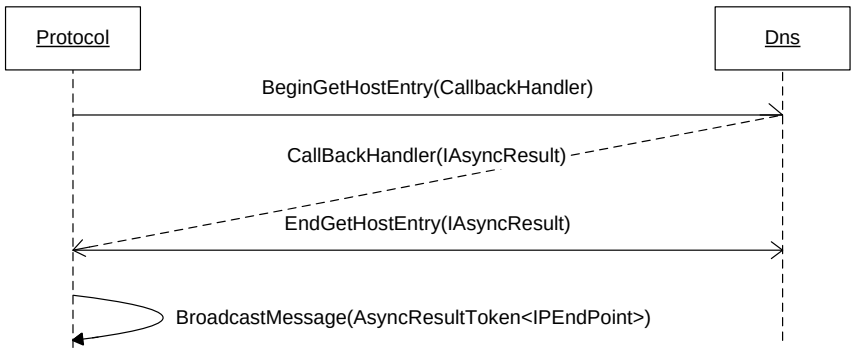

Abbildung 3.10: Sequenzdiagramm eines asynchronen Methodenaufrufs

Ein *AsyncResultReceiver* wartet auf den Eingang einer Nachricht vom Typ *AsyncResultToken<T>*. Diese von *MessageBase* erbende Klasse stellt die drei Eigenschaftsfelder *Protocol*, *Completed* und *Result* bereit. Das erste Eigenschaftsfeld beinhaltet die Information, in welchem Protokoll die Nachricht verarbeitet werden soll. Das zweite Property gibt an, ob die asynchrone Methode terminiert ist oder ob das Ergebnis noch aussteht. Das letzte, mit dem Typparameter T typisierte Eigenschaftsfeld *Result* enthält das Ergebnis des asynchronen Aufrufs.

Der *AsyncResultReceiver* kapselt den asynchronen Methodenaufruf. Der *Receiver* wartet auf den *Callback*, welcher noch im falschen *Thread* aufgerufen wird, und verpackt das Ergebnis in einer typisierten *AsyncResultToken<T>*-Nachricht. Diese wird im Anschluss in die Warteschlange des Gears4Net Protokolls eingestellt und kann dann, wie im obigen Abschnitt beschrieben, von den *AsyncResultReceivern* weiterverarbeitet werden.

Das beschriebene Szenario ist analog zur Darstellung des asynchronen Aufrufs im Sequenzdiagramm 3.11 dargestellt und verdeutlicht die fünf, für einen asynchronen Aufruf notwendigen Arbeitsschritte. Das Diagramm verdeutlicht, dass der Entwickler des aufrufenden *Protocols* lediglich die Initialisierung des *Receivers* sowie den Aufruf der *Begin...*-Methode initiieren muss. Die folgenden Arbeitsschritte werden automatisiert vom *AsyncResultReceiver* übernommen.

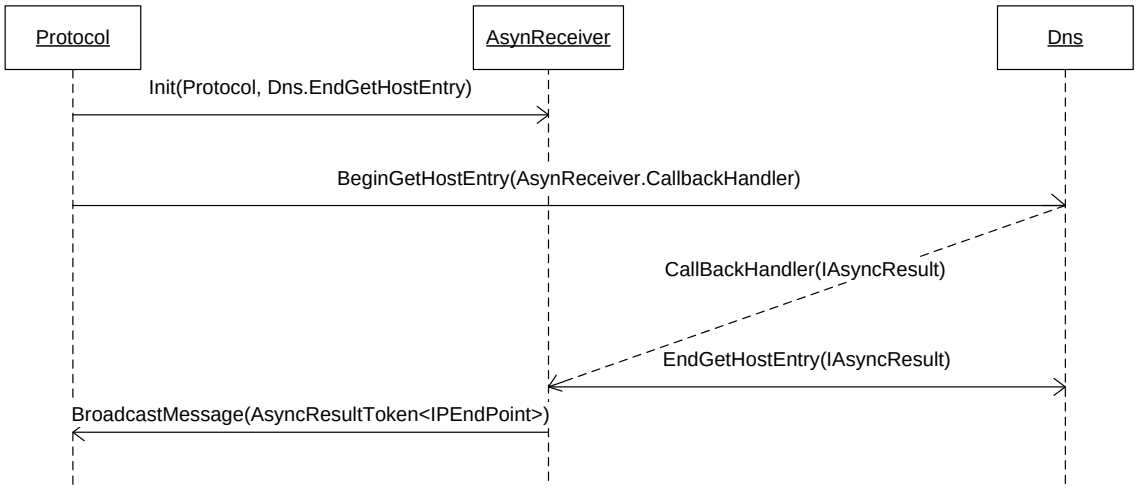

Abbildung 3.11: Asynchroner Methodenaufruf mittels eines AsyncResultReceivers

AsyncResultReceiver<T>

Der *AsyncResultReceiver<T>* findet seine Anwendung bei asynchronen Methodenaufrufen, die kein Status-Objekt beim Aufruf der *Begin...*-Methode erwarten. Die Verwendung des *AsyncResultReceiver<T>* wird in Auflistung 3.14 anhand eines exemplarischen Aufrufs der Methode *BeginGetHostEntry* dargestellt. Das Methodentupel *BeginGetHostEntry* und *EndGetHostEntry* liefert ein Resultat vom Typ *IPHostEntry*. Der *AsyncResultReceiver<T>* wird daher in Zeile 1 der Auflistung mit diesem Typ typisiert. Im Konstruktor des Receivers wird neben dem aktuellen Protokoll auch ein Zeiger auf die *EndGetHostEntry* übergeben. In der zweiten Zeile der Auflistung wird der asynchrone Methodenaufruf *BeginGetHostEntry* durchgeführt. Hierbei ist zu beachten, dass der als zweites übergebene Funktionszeiger nicht wie üblich auf eine Funktion innerhalb der umschließenden Klasse zeigt, sondern auf eine im *AsyncResultReceiver<T>* definierte Funktion namens *AsyncResultHandler*. Abschließend wird mittels einer *yield*-Anweisung auf den Eingang der Nachricht *AsyncResultToken<IPHostEntry>* gewartet, die das Ergebnis des Methodenaufrufs im *Result*-Property bereitstellt.

```
1       AsyncResultReceiver<IPHostEntry> receiver = new AsyncResultReceiver<
            IPHostEntry>(this, Dns.EndGetHostEntry);
2       Dns.BeginGetHostEntry("localhost", receiver.AsyncResultHandler, null);
3
4       yield return receiver;
5       IPHostEntry myEntry = receiver.Result;
```

Auflistung 3.14: Verwendung des AsyncResultReceivers

Die Signatur der *AsyncResultReceiver<T>*-Klasse ist im Klassendiagramm in Abbildung 3.12 dargestellt. Der *Receiver* verfügt über zwei Konstruktoren, wobei einer als privater *Copy*-Konstruktor und der andere als öffentliche Variante bereitgestellt wird. Der öffentliche Konstruktor verfügt über eine Parameterliste mit zwei Elementen. Der erste Parameter bezeichnet das Protokoll, in dessen Warteschlange das Resultat des asynchronen Methodenaufrufs abgelegt werden soll. Der zweite Parameter vom Typ *EndCallbackHandler* erwartet einen Zeiger auf die *End...*-Methode, die aufgerufen werden soll, um das Resultat des asynchronen Aufrufs abzufragen. Der *EndCallbackHandler* ist ein generischer Delegant, der ein *IAsyncResult* als Parameter erwartet und mit einem beliebigen Typparameter *T* als Rückgabe typisiert werden kann. Im Falle des oben aufgeführten Beispiels handelt es sich um die Methode *DNS.EndGetHostEntry* mit dem Rückgabetyp *IPHostEntry*.

Die von jeder asynchronen *Begin...*-Methode erwartete *Callback*-Methode, die der Signatur des *System.AsyncCallback*-Deleganten entspricht, wird als öffentliche *AsyncResultHandler*-Methode innerhalb der *AsyncResultReceiver<T>*-Klasse implementiert. Sobald die *Callback*-Methode aufgerufen wird, prüft diese, ob der im Konstruktor übergebene Zeiger auf eine valide Funktion zeigt. Ist dies der Fall, so wird die Funktion über den Zeiger aufgerufen und das Ergebnis des Aufrufs im Eigenschaftsfeld *Result* gespeichert. Der asynchrone Methodenaufruf ist somit beendet. Das Resultat wird nun als Nachricht verpackt und über die Warteschlange des im Konstruktor übergebenen Protokolls verbreitet.

Bei dem aufgezeigten Verfahren wird die asynchrone Methode aus dem *Scheduler-Thread* der aktuellen *Protocol*-Instanz angerufen. Der Rückruf des *Callback-Handlers* erfolgt im Kontext eines beliebiges *Threads* aus dem vom Betriebssystem bereitgestellten *Thread-Pool*. Die *Gears4Net*-Nachricht, die das Ergebnis des asynchronen Methodenaufrufs enthält, wird somit im Kontext des *Thread-Pool-Threads* erzeugt und in die Warteschlange der *Protocol*-Instanz eingestellt. Die Verarbeitung der Nachricht erfolgt dann gemäß des *Gears4Net*-Modells im Kontext des *Scheduler-Threads* der *Protocol*-Instanz.

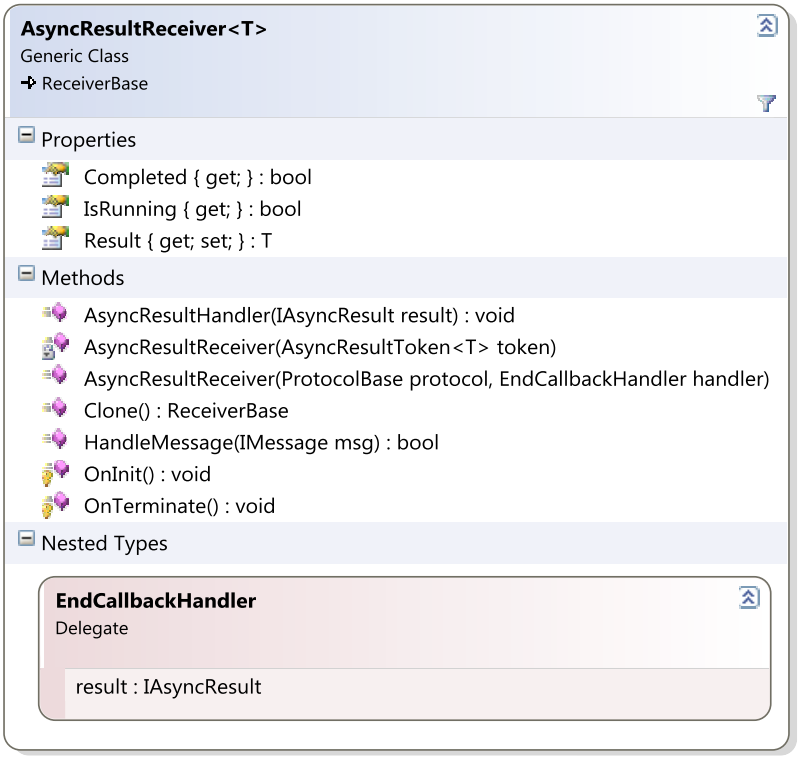

Abbildung 3.12: Klassendiagramm der AsyncResultReceiver-Klasse

AsyncResultReceiver<T, O>

Die Funktionsweise des *AsyncResultReceivers* mit zwei generischen Typisierungsparametern ist analog zu dem im vorherigen Abschnitt beschriebenen *Receiver* zu betrachten. Der *AsyncResultReceiver<T, O>* verfügt im Gegensatz zu der Variante mit nur einem Typisierungsparameter über ein zusätzliches Status-Objekt, das beim Aufruf der *Begin...*-Methode mit übergeben werden kann. Die Übergabe von Status-Objekten ist bei der Verwendung der asynchronen .NET Syntax üblich und findet beispielsweise bei einem asynchronen Lesen auf einem *Socket* Anwendung.

3.6.4 State-Machine

Die Verbindung des Konzeptes einer Zustandsmaschine mit der Effizienz der asynchronen Programmierung bildet das Herzstück der Gears4Net Idee. Die Vereinigung dieser beiden Konzepte wirkt auf den ersten Blick konträr. Während das erste Modell auf die Verwendung von blockierenden Systemaufrufen setzt, fordert die asynchrone Programmierung die Benachrichtigung durch *Callback*-Methoden.

Weicht man die Forderung des Zustandsmaschinenmodells dahingehend auf, dass vollständig auf die Verwendung von blockierenden Systemaufrufen zugunsten von

speziellen Wartebedingungen verzichtet werden kann, bietet dieses zusammen mit einem erweiterten Ausführungsumgebung ein neues Programmierparadigma. Die Ausführungsumgebung führt den Quellcode einer Zustandsmaschine solange aus, bis diese auf eine Wartebedingung stößt. Ist die Wartebedingung instantan erfüllt, so wird mit der Ausführung fortgefahren. Ist dies nicht der Fall, wird die Methode unterbrochen und erst bei späterer Erfüllung der Bedingung an gleicher Stelle fortgeführt. Der ausführende *Thread* wird der ausgeführten Methode zum Zeitpunkt der Unterbrechung entzogen und zu einem späteren Zeitpunkt zum Fortführen der Methode wieder zugewiesen.

Die Umsetzung dieses Konzeptes erfordert neben den in den Kapiteln 3.6.3 und 3.6.6 dargestellten Wartebedingungen und Schedulingverfahren die im Fokus dieses Abschnitts stehenden unterbrechbaren Funktionen. Betrachtet man die Ausführung eines Unterprogramms, wird deutlich, dass eine Unterbrechung einer Methode einen tiefen Eingriff in die Ausführungsumgebung und das darunterliegende Framework bedeutet.

3.6.4.1 Unterbrechbare Methoden

Die Unterbrechung einer Methode, Funktion oder eines beliebigen Unterprogramms und der anschließende Wechsel zu einer anderen Funktion erweist sich als äußerst schwierig, da es die Sicherung und spätere Wiederherstellung des aktuellen *Stacks* inklusive der *Stack-* und *Basepointer* implizieren würde. Dieses Verfahren erfordert einen tiefgehenden Eingriff in die Ausführungsumgebung und wirft zudem Effizienz- und Konsistenzfragen auf.

Der Sicherungs- und Wiederherstellungsmechanismus bedingt das kopieren der Daten des *Stacks*. Bei einer durchschnittlichen Größe des *Stacks* von beispielsweise *512 KB* würde ein solches Verfahren einen Datentransfer von mindestens *1 MB* zur Folge haben. Eine Web-Serveranwendung, die beispielsweise *1000 Clients* versorgt und für diese jeweils eine unterbrochene Kommunikationsmethode sowie eine ebenfalls wartende *I/O*-Methode im Speicher hält, verbraucht ausschließlich für die Sicherung der *Threads 1000 * (512 KB + 512 KB) = 1 GB* Arbeitsspeicher.

Die Ineffizienz dieses Verfahrens wird zusätzlich noch von der Konsistenzfrage begleitet. Ein wiederhergestellter *Stack* ist konsistent, solange dieser entweder keine Referenzen auf ein Objekt auf dem *Heap* enthält oder die vorhandenen Referenzen noch gültig sind. Es ist beispielsweise möglich, dass Objekte, die von einer Varia-

blen des *Stacks* referenziert werden, durch den *Garbage Collector* aus dem Speicher entfernt wurden, während der *Stack* als Datenblock gesichert ist.

Die vorgestellten Konsistenzrisiken mit den einhergehenden Effizienzproblemen erfordern weitergehende Mechanismen zur Unterbrechung von Methoden und Funktionen. Einen solchen Mechanismus stellt das aus der Programmiersprache Python [122] bekannte und in Kapitel 2.6 dargestellt Iteratorenkonzept des Microsoft .NET Frameworks im Zusammenspiel mit der *yield-return*-Anweisung bereit, das im Folgenden vorgestellt wird.

Nutzung des Iteratorenkonzeptes

Die Methodenblöcke des Iteratorenkonzeptes bilden die Grundlage für pseudo-unterbrechbare Methoden im Microsoft .NET Framework. Die Begrifflichkeit der *pseudo unterbrechbaren Methode* ist durch die Umsetzung des Iteratorenkonzeptes bedingt. Zur Entwicklungszeit stellt sich ein Iterator als Methode dar, die durch die Rückgabetypen *IEnumerator*, *IEnumerable* beziehungsweise ihren generischen Äquivalenten charakterisiert ist und zudem Unterbrechungen in Form von *yield-return*-Anweisungen zulässt. Der Betrachtungswinkel ändert sich mit der Kompilierung. Der C#-Compiler wandelt den Iteratorblock in eine Klasse um, wobei jede *yield-return*-Anweisung des Blockes zu einer Fallunterscheidung in der *MoveNext*-Methode der neuen Klasse führt.

```
1   public IEnumerator<long> Fibonacci()
2   {
3       yield return 0;
4       yield return 1;
5
6       long pred1 = 1, pred2 = 0;
7
8       while (true)
9       {
10          long value = pred1 + pred2;
11          pred2 = pred1;
12          yield return pred1 = value;
13      }
14  }
```

Auflistung 3.15: Implementierung der Fibonacci Funktion in einem Iterator

Der Überführungsprozess eines Iteratorblocks zu einer Iteratorklasse wird im Folgenden anhand der Implementierung der Fibonacci-Zahlen aufgezeigt. Der *Fibonacci*-Iterator in Auflistung 3.15 erzeugt bei jedem Aufruf das jeweils nächste Element der Fibonacci-Folge. Ein Element ist genau dann produziert, wenn es mittels der

yield-return-Anweisung zurückgegeben wird. Im Fall des Beispiels in Auflistung 3.15 liefert der Algorithmus in den Zeilen 3 und 4 jeweils die initialen Werte 0 und 1 und beginnt anschießend mit der Berechnung weiterer Elemente, die in Zeile 12 durch die *yield-return*-Anweisung ausgegeben werden.

```
1       IEnumerator<long> enumerator = Fibonacci();
2       while (enumerator.MoveNext())
3       {
4           Console.WriteLine(enumerator.Current);
5       }
```

Auflistung 3.16: Expliziter Aufruf des Fibonacci Iterators

Der explizite in Auflistung 3.16 dargestellte Aufruf eines Iterator verrät im Gegensatz zum impliziten Verfahren (siehe Kapitel 2.6) die duale Betrachtungsweise des Iteratorkonzeptes. Die Traversion über den *Fibonacci*-Iterator erfolgt über das Zusammenspiel der *MoveNext*-Methode und des *Current*-Properties des Enumerators und steht somit nicht im direkten Zusammenhang mit der in Auflistung 3.15 dargestellten Implementierung des Iterator selbst. Der Zusammenhang verdeutlicht sich erst bei der Analyse der vom Compiler generierten und in der Microsoft .NET Assembly abgelegten Iteratorklasse. Ein Auszug der generierten Fibonacci-Iteratorklasse ist in Auflistung 3.17 dargestellt. Jede Iteratorklasse verfügt über eine *Current*-Property, über das das aktuell produzierte Element abgefragt werden kann, und eine *MoveNext*-Methode, die für die Erzeugung der Elemente zuständig ist. Letztere Methode von booleanschem Rückgabetyp evaluiert *wahr*, falls noch weitere Elemente produziert werden können, und andernfalls *falsch*. Die angesprochene Überführung der Funktionalität des Iteratorblocks in die *MoveNext*-Methode der Iteratorklasse erfolgt durch die Implementierung einer Zustandsmaschine. Innerhalb der *MoveNext*-Methode wird dazu eine Fallunterscheidung in Form einer *switch*-Anweisung implementiert, die auf einer klassenweiten Zustandsvariable agiert.

Im vorliegenden Beispiel ist die Zustandsvariable *State* vor dem ersten Aufruf der *MoveNext*-Methode mit dem Wert *0* initialisiert. Durch diese Konfiguration springt die *switch*-Anweisung in die erste Verzweigung und setzt den Wert des *Current*-Properties auf *0* sowie die Zustandsvariable auf *1*. Ein zweiter Aufruf von *MoveNext* führt in der veränderten Konfiguration zum Sprung in die zweite Verzweigung. Das Verfahren wird solange fortgeführt, bis die Zustandsvariable den Wert *3* angenommen hat. In dieser Konfiguration springt der *Program-Counter* aufgrund der *break*-Anweisung in Zeile 27 aus der *switch*-Anweisung heraus und setzt die Berechnung eines weiteren Elementes in Zeile 32 fort. Der vorliegende Iterator wird niemals selb-

```
1       public int State  { get; private set; }
2       public long Current { get; private set; }
3
4       private long pred1;
5       private long pred2;
6
7       private bool MoveNext()
8       {
9           switch (this.State)
10          {
11              case 0:
12                  this.Current = 0;
13                  this.State = 1;
14                  return true;
15
16              case 1:
17                  this.Current = 1;
18                  this.State = 2;
19                  return true;
20
21              case 2:
22                  this.pred1 = 1;
23                  this.pred2 = 0;
24                  break;
25
26              case 3:
27                  break;
28
29              default:
30                  return false;
31          }
32          long value = this.pred1 + this.pred2;
33          this.pred2 = this.pred1;
34          this.Current = this.pred1 = value;
35          return true;
36      }
```

Auflistung 3.17: Auszug aus der generierten Fibonacci Iterator Klasse

ständig terminieren, da die *default*-Sprungmarke in Zeile 29 nicht angesprungen wird und der darin enthaltene Quellcode deshalb nicht ausgeführt werden kann.

Das vorgestellte Iteratorenkonzept eignet sich aufgrund der beschriebenen Dualität ideal für die Implementierung von unterbrechbaren Methoden. Die Syntax der Iteratorblöcke ermöglicht eine einfache Implementierung und bietet dem Entwickler zudem durch das Schlüsselwort *yield-return* ein intuitives Verständnis für die Unterbrechungssemantik. Vervollständigt wird das Konzept durch die Generierung der Iteratorklassen. Dieses Verhalten erlaubt die höchstmögliche Effizienz, da ein Iterator zur Laufzeit nichts weiter als die Instanz einer Klasse darstellt und somit effektiv und effizient verwaltet werden kann. Weitere Effizienzbetrachtungen des Ite-

ratorkonzeptes sowie der Implementierung der *State-Machines* erfolgen im Anschluss in Kapitel 3.8.1.

3.6.4.2 Implementierung der StateMachine-Klassen

Die Vereinigung von unterbrechbaren Methoden und Wartebedingungen bildet das Herzstück des Gears4Net Konzeptes. Die Symbiose des Iteratorenkonzeptes mit den in Kapitel 3.6.3 vorgestellten *Receivern* findet ihre Implementierung in den *State-Machine*-Klassen.

Das Gears4Net Framework stellt vier *StateMachine*-Klassen bereit, die sich ausschließlich in der Anzahl der verwalteten Parameter unterscheiden, die an einen Iterator übergeben werden können. Die funktionalen Gemeinsamkeiten der vier Klassen werden daher aus den Klassen heraus faktorisiert und in der gemeinsamen abstrakten Basisklasse *AbstractStateMachine* abgelegt.

Das Modell der Zustandsmaschinen ist im Gears4Net Framework aus zwei Blickwinkeln zu betrachten. Aus der Sicht eines Entwicklers, dessen Applikationen auf dem Framework basieren, stellt sich eine Zustandsmaschine als ein Iterator mit Rückgabetyp *ReceiverBase* dar, in dem Wartebedingungen in Form von *Receivern* spezifiziert werden können. Aus Sicht des Gears4Net Frameworks besteht ein Automat aus einer von *AbstractStateMachine* erbenden Klasse, die einen Funktionszeiger auf den Iterator besitzt, der den Quellcode des Entwicklers enthält.

Die maximale Transparenz für den Applikationsentwickler ist einer der primären Zielsetzungen bei der Gears4Net Idee. Dies spiegelt sich in der Sichtweise des Entwicklers auf die Zustandsmaschinen wider und ist in Auflistung 3.18 dargestellt. In dem aufgezeigten Beispiel verfügt der *Execute*-Iterator über eine Parameterliste mit nur einem Parameter, bei dem es sich um die Instanz einer von *AbstractStateMachine* erbenden Klasse handelt. Die Wahl der *StateMachine*-Klasse ist abhängig von der Anzahl der Parameter, die zusätzlich an den Iterator übergeben werden können.

```
1      public IEnumerator<ReceiverBase> Execute(AbstractStateMachine stateMachine)
2      {
3          // Add Custom Code here ...
4          yield return Receive<PingMessage>();
5      }
```

Auflistung 3.18: Iterator einer AbstractStateMachine mit drei Parametern

Mit Abschluss der Deklaration eines Iterators kann sich der Entwickler vollständig auf die Implementierung der Automatenlogik konzentrieren. Im Fall des aufgezeigten Beispiels beinhaltet dies die Spezifikation einer *Receiver*-Wartebedingung, die auf eine Nachricht des Typs *PingMessage* wartet.

Die AbstractStateMachine-Klasse

Die Implementierung der Zustandsmaschinen des *Gears4Net*-Frameworks erfolgt über die von der Klasse *AbstractStateMachine* erbenden Klasse *StateMachine*. Zu den Aufgaben der *StateMachine*-Klassen gehören neben der Verwaltung des Iterators auch das Persistieren des Automatenzustandes, die Zustellung von Nachrichten, die Verwaltung der *Receiver*-Wartebedingungen sowie die Überwachung der Terminierungsbedingungen.

Das in Abbildung 3.13 dargestellte Klassendiagramm der *AbstractStateMachine* zeigt die für die einzelnen Teilbereiche relevanten Felder, Properties und Methoden. Die Initialisierung der Zustandsmaschine beginnt mit dem Aufruf der *Init*-Methode, die ein Protokoll als Referenzparameter erwartet und den Status des Automaten auf den initialen Wert *StateMachineStatus.Waiting* festlegt. Im Laufe des Lebenszyklus eines Automaten kann dieser noch die Zustände *StateMachineStatus.Running* und *StateMachineStatus.Terminated* einnehmen.

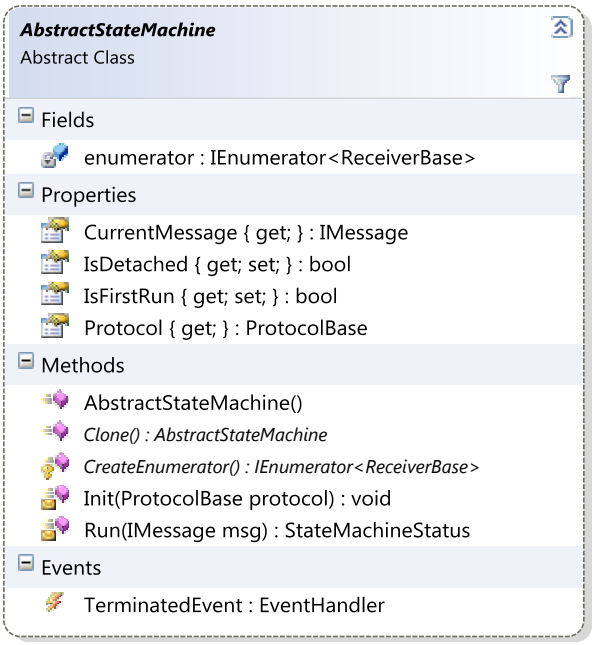

Abbildung 3.13: Klassendiagramm der AbstractStateMachine-Klasse

Der zweite Bereich, die Verwaltung der Zustandsmaschinen erfolgt über das private Feld *enumerator* sowie die abstrakte Methode *CreateEnumerator*, die von jeder Klasse implementiert werden muss, die von *AbstractStateMachine* erbt. Die Erzeugung eines Iterators kann aus Flexibilitätsgründen nicht direkt in der *AbstractStateMachine*-Klasse vorgenommen werden, da das Framework Iteratoren mit einer unterschiedlichen Anzahl an Parametern zulassen kann. Das während der Deklaration als *IEnumerator<ReceiverBase>* typisierte *enumerator*-Feld speichert die Referenz auf die Instanz des in der *CreateEnumerator*-Methode erzeugten Iterator-Objektes, die im Zuge des Initialisierungsprozesses aufgerufen wird. Mit Hilfe des *enumerator*-Feldes ist es nun möglich, einen Iterator über die *MoveNext*-Methode auszuführen, bis eine *yield-return*-Anweisung auftritt. Die im *Current*-Property des *enumerators* abgelegte *Receiver*-Wartebedingung kann anschließend von der *AbstractStateMachine* ausgelesen und verarbeitet werden.

```
1       public IEnumerator<ReceiverBase> Execute(AbstractStateMachine stateMachine)
2       {
3           Foo();
4           yield return Receive<A>();
5
6           Bar();
7           yield return Receive<B>();
8       }
```

Auflistung 3.19: Nachrichtenverarbeitungsschritte

Die Zustellung von Nachrichten und die Verarbeitung der *Receiver*-Wartebedingungen bilden den dritten Aufgabenbereich der *AbstractStateMachine*-Klasse. Die Arbeitsschritte der Nachrichtenverarbeitung werden vorweg anhand des Beispielquellcodes in Auflistung 3.19 dargelegt. Eine Zustandsmaschine verfügt im Moment ihrer Erzeugung über keine aktuelle Wartebedingung, da diese erst mit dem ersten Aufruf der *yield-return*-Anweisung gesetzt wird. Abbildung 3.19 verdeutlicht dieses anhand der Aufrufreihenfolge. Der erste *yield-return*-Aufruf erfolgt in Zeile 4. Bis dahin ist das *Current*-Property des Iterator mit dem initialen Wert *null* belegt. Sobald die *yield-return*-Anweisung in Zeile 4 aufgerufen wird, werden dem zurückgelieferten *Receiver*, der auf Nachrichten des Typs *A* wartet, eingehende Nachrichten zugestellt. Mit der Zustellung wird solange fortgefahren, bis die Wartebedingung erfüllt ist. Die Ausführung wird anschließend durch den Aufruf der *enumerator.MoveNext*-Methode fortgesetzt. Dieses führt zur Ausführung der Methode *Bar* und schließlich zu einer weiteren *yield-return*-Anweisung, nach deren Erfüllung die Zustandsmaschine terminiert.

```
1       internal StateMachineStatus Run(IMessage msg)
2       {
3           this.status = StateMachineStatus.Running;
4           this.msg = msg;
5
6           bool continueExecution = true;
7
8           if (HandleMessage(this.receiver, this.msg))
9           {
10              do
11              {
12                  continueExecution = this.enumerator.MoveNext();
13                  if (continueExecution && this.enumerator.Current != null)
14                  {
15                      this.receiver = this.enumerator.Current;
16                      this.receiver.Init(this.protocol);
17                  }
18                  else
19                  {
20                      this.receiver = null;
21                      continueExecution = false;
22                  }
23              }
24              while (this.receiver != null && this.receiver.IsTerminated &&
                        !(this.msg is TerminateMessage));
25          }
26
27          if (this.msg is TerminateMessage || !continueExecution)
28          {
29              this.status = StateMachineStatus.Terminated;
30              if (this.receiver != null)
31              {
32                  this.receiver.Terminate();
33                  this.receiver = null;
34              }
35
36              this.enumerator = null;
37
38              if (this.IsDetached)
39                  this.Protocol.RemoveDetachedStateMachine(this);
40
41              if (this.TerminatedEvent != null)
42                  this.TerminatedEvent(this, EventArgs.Empty);
43          }
44          else
45          {
46              this.status = StateMachineStatus.Waiting;
47          }
48
49          this.msg = null;
50          return this.status;
51      }
```

Auflistung 3.20: Nachrichtenzustellung an einen Receiver innerhalb der Run-Methode

Der präzise Ablauf des gerade skizzierten Verfahrens ist aus der Implementierung, der für die Nachrichtenverarbeitung zuständigen *Run*-Methode ersichtlich. Jeder Aufruf der in Auflistung 3.20 dargestellten Methode überführt die *StateMachine* zuerst in den Statuszustand *StateMachineStatus.Running* und speichert die zu verarbeitende Nachricht des Typs *IMessage*, die als Parameter übergeben wurde, im Klassenfeld *msg*. Die Verarbeitung der eingegangenen Nachricht erfolgt durch den in die *if*-Anweisung verschachtelten Aufruf der *HandleMessage*-Methode in Zeile 8. Die *HandleMessage*-Methode, deren Implementierung in Auflistung 3.21 aufgeschlüsselt ist, stellt die Nachricht dem aktuellen *Receiver* zu, auf den über das Klassenfeld *receiver* zugegriffen werden kann. Selbige Methode liefert über ihren booleanschen Rückgabewert die Information, ob die Wartebedingung des *Receivers*, dem die Nachricht zugestellt wurde, erfüllt ist oder nicht. Beim ersten Aufruf der *Run*- beziehungsweise der *HandleMessage*-Methode ist das *Receiver*-Feld mit *null* initialisiert. In diesem Spezialfall evaluiert die *HandleMessage*-Methode ebenfalls den Wert *wahr*, da mit der weiteren Verarbeitung des *StateMachine*-Quellcodes fortgefahren werden kann.

```
1       private bool HandleMessage(ReceiverBase receiver, IMessage msg)
2       {
3           bool complete = true;
4           if (receiver != null)
5           {
6               complete = receiver.HandleMessage(msg);
7               if (complete)
8               {
9                   receiver.Terminate();
10                  receiver = null;
11              }
12          }
13          return complete;
14      }
```

Auflistung 3.21: HandleMessage-Methode

Insofern kein aktiver *Receiver* definiert ist, der auf die Erfüllung seiner Bedingung wartet, wird der Quellcode der *do-while*-Schleifen in den Zeilen 10 bis 24 ausgeführt. Die Schleife wird solange durchlaufen, bis die Schleifenabbruchbedingung in Zeile 24 erfüllt ist. Diese besagt, dass die Schleife solange durchlaufen werden soll, bis ein nicht terminierter *Receiver* gefunden wurde, der Iterator zu Ende gelaufen ist oder eine *TerminateMessage* zugestellt wurde. Im Schleifenbauch wird die schrittweise Verarbeitung des Iterator Quelltextes vorgenommen. Der Aufruf der *enumerator.MoveNext()*-Methode in Zeile 12 führt den Quelltext zwischen der letzten Unterbrechung und der nächsten Wartebedingung aus. Der Rückgabewert der

Methode, der in der Variablen *continueExecution* gespeichert wird, gibt an, ob der Iterator bereits zu Ende gelaufen ist oder ob noch weitere Verarbeitungsschritte nötig sind. Der aktuelle *Receiver* wird im Klassenfeld *receiver* gespeichert und mittels des Aufrufs der *Init*-Methode in Zeile 16 initialisiert. Für den Fall, dass der Iterator zu Ende gelaufen ist oder fälschlicher Weise anstatt eines Objektes des Typs *ReceiverBase* ein *null*-Wert im Iterator zurückgeliefert wurde, springt die Ausführung in den *else*-Zweig in Zeile 18. Dieser setzt den aktuellen *receiver* auf den Wert *null* und unterbindet die weitere Ausführung des Iterators, indem die *continueExecution*-Variable auf den Wert *falsch* gesetzt wird.

Im Anschluss an die Zustellung der eingegangenen Nachricht an den aktiven *Receiver* und die gegebenenfalls ausgeführte *do-while*-Schleife wird mit der Ausführung des Quelltextes in Zeile 27 fortgefahren. Die *if*-Anweisung entscheidet an dieser Stelle, ob die Terminierungsbedingung für die *StateMachine* erfüllt ist oder nicht. Für den Fall, dass keine *TerminateMessage* eingegangen ist und die Variable *continueExecution* nicht auf *falsch* steht, wird der Status des Automaten in Zeile 43 auf *StateMachineStatus.Waiting* gesetzt. Die Maschine ist damit bereit, neue Nachrichten zu verarbeiten. Andernfalls wird in den in Zeile 28 beginnenden Block eingetreten und mit der Terminierung begonnen. Der Status der Maschine wird dazu auf *StateMachineStatus.Terminated* gesetzt. Darüber hinaus wird der aktuelle *Receiver* terminiert und die Referenz auf den Iterator annulliert. Sollte es sich bei der *StateMachine* um einen Automaten gehandelt haben, der mit dem Flag *detached* gestartet wurde, so wird die Machine zusätzlich aus der Liste der *DetachedStateMachines* des aktuellen Protokolls entfernt. Die Terminierung einer Zustandsmaschine wird mit dem Aufruf des *TerminateEvent-Handler* abgeschlossen. Diese Benachrichtigung kann sowohl von einem Protokoll als auch von einem *StateMachineReceiver* aufgefangen und entsprechend weiterverarbeitet werden.

Der Abschluss der *Run*-Methode der *AbstractStateMachine*-Klasse erfolgt durch die Rückgabe des aktuellen Status sowie die Vernichtung der Referenz auf die aktuelle bearbeitete Nachricht.

Die StateMachine-Klasse

Das Gears4Net Framework stellt neben der abstrakten Basisklasse *AbstractStateMachine* vier *StateMachine*-Klassenimplementierungen bereit. Die einzelnen Klassen unterscheiden sich, wie bereits angesprochen, in der Anzahl der Parameter, die an den Iteratorblock übergeben werden können. Die Bandbreite reicht dabei von Iteratoren mit keinem bis hin zu drei generisch typisierten Parametern. Aufgrund

der ähnlichen Klassenstruktur konzentriert sich die Beschreibung auf eine exemplarisch ausgewählte Klasse, in diesem Fall mit drei generischen Parametern, deren Klassendiagramm in Abbildung 3.14 dargestellt ist.

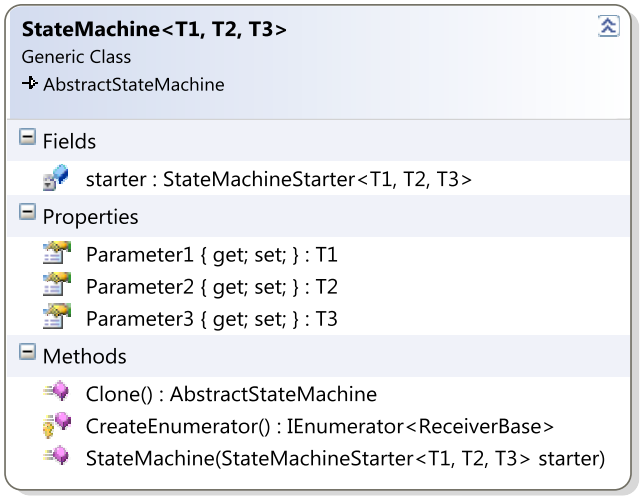

Abbildung 3.14: Klassendiagramm der StateMachine-Klasse mit drei generischen Parametern

Die Klasse verfügt über drei Properties, die jeweils über die Typparameter *T1*, *T2* sowie *T3* während der Deklaration der Klasse typisiert werden können. Darüber hinaus stellt die Klasse die beiden von *AbstractStateMachine* vererbten Methoden *CreateEnumerator* und *Clone* sowie einen Konstruktor bereit, der als Parameter einen typsicheren Funktionszeiger erwartet. Bei diesem Zeiger handelt es sich um einen Deleganten, der auf den zur *StateMachine* zugehörigen Iterator zeigt. Aufgrund der strengen Typsicherheit der Deleganten muss die Signatur des Iterators mit dem des Funktionszeigers übereinstimmen. Der übergebene Funktionszeiger wird im privaten Feld *starter* abgelegt und wird für die Erzeugung eines neuen Enumerators innerhalb der *CreateEnumerator*-Methode sowie zur Erstellung von duplizierten Zustandsmaschinen in der *Clone*-Methode verwandt.

Die mit dem Schlüsselwort *override* gekennzeichnete Methode *CreateEnumerator* in Auflistung 3.22 ruft die hinter dem Funktionszeiger *starter* verborgene Iteratormethode auf und übergibt neben den drei generisch typisierten Parametern auch eine Referenz auf die aktuelle Instanz der *StateMachine*. Die Aufgabe der *Clone*-Methode liegt in der Erzeugung einer neuen Zustandsmaschine, wobei diese auf die exakt gleiche Iteratormethode zeigt und die identischen Parameter übergibt.

```
1       protected override IEnumerator<ReceiverBase> CreateEnumerator()
2       {
3           return this.starter(this, this.Parameter1, this.Parameter2, this.
                Parameter3);
4       }
5
6       public override AbstractStateMachine Clone()
7       {
8           return new StateMachine<T1, T2, T3>(this.starter) { Parameter1 = this.
                Parameter1, Parameter2 = this.Parameter2, Parameter3 = this.
                Parameter3 };
9       }
```

Auflistung 3.22: CreateEnumerator- und Clone-Methode der StateMachine-Klasse

```
1       public class StateMachineCreator
2       {
3           public void CreateCustomStateMachine()
4           {
5               StateMachine<IPAddress, int, ProtocolType> myStateMachine = new
                    StateMachine<IPAddress, int, ProtocolType>(CustomSMIterator);
6               myStateMachine.Parameter1 = IPAddress.Parse("127.0.0.1");
7               myStateMachine.Parameter2 = 80;
8               myStateMachine.Parameter3 = ProtocolType.Tcp;
9           }
10
11          public IEnumerator<ReceiverBase> CustomSMIterator(AbstractStateMachine
                stateMachine, IPAddress param1, int param2, ProtocolType param3)
12          {
13              // Add custom code
14          }
15      }
```

Auflistung 3.23: Erstellung einer StateMachine

3.6.5 Die ProtocolBase-Klasse

Das Gears4Net Framework orientiert sich am Aktor basierten Programmiermodell [3, 4, 25, 52] und erweitert dieses um das Modell der Zustandsmaschinen ähnlichen Programmierung in einer asynchronen Umgebung. Der Lebenszyklus eines solchen Aktors erstreckt sich von der Erzeugung und Initialisierung über die Verarbeitung von Nachrichten bis hin zu seiner Terminierung. Der komplette Lebenszyklus wird dabei von der Ausführungsbedingung eines Aktors begleitet, die besagt, dass jeder Aktor von einer eigenen Recheneinheit und dediziertem Speicher ausgeführt wird und zudem über asynchrone Nachrichten mit anderen Aktoren kommuniziert.

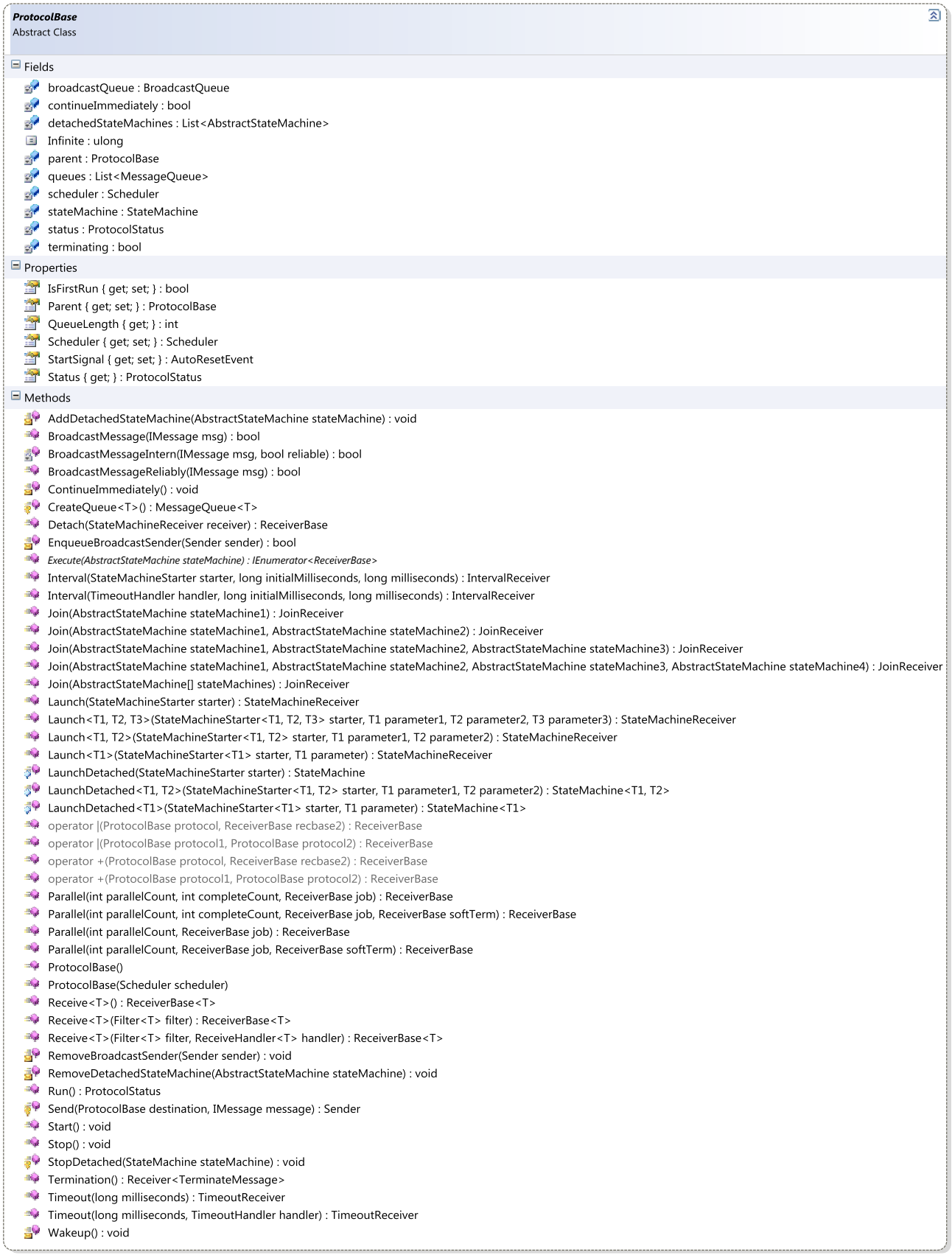

Abbildung 3.15: Klassendiagramm der ProtocolBase-Klasse

Die skizzierte Anforderungsmenge findet ihre Implementierung in der in Abbildung 3.15 dargestellten abstrakten Basisklasse *ProtocolBase*, von der jedes benutzerdefinierte *Protocol*, das im Kontext des Gears4Net Frameworks ausgeführt wird, erben muss. Die *ProtocolBase* stellt mit dem *Execute*-Iterator den initialen Einstiegspunkt in die Welt der Zustandsmaschinen bereit. Zudem verfügt die Klasse über die *broadcastQueue*-Warteschlange, über die die Zustandsmaschinen mit Nachrichten versorgt werden. Darüber hinaus beinhaltet die *ProtocolBase*-Klasse eine Vielzahl an kleinen Hilfsfunktionen, die die Entwicklung von benutzerdefinierten *Protocol*-Klassen vereinfachen. Zu diesen zählen beispielsweise Methoden zum Hinzufügen von Nachrichten in die Warteschlange oder weitere überladene Operatoren, die die Syntax der Gears4Net Applikationen vereinfachen können. Abschließend enthält die *ProtocolBase*-Klasse neu eingeführte Kommandos, die den Umgang mit *Receiver*-Wartebedingungen extrem vereinfachen und die Lesbarkeit des Quelltextes signifikant erhöhen.

Protocol-Lebenszyklus

Der Lebenszyklus eines *Protocols* beginnt mit seiner Instanziierung. Die *ProtocolBase*-Klasse stellt dazu einen Standardkonstruktor bereit, der die angesprochene *broadcastQueue* initialisiert, und einen Konstruktor, der als Parameter die Referenz auf einen *Scheduler* erwartet, zu dem die aktuelle *Protocol*-Instanz hinzugefügt wird. Die übergebene *Scheduler*-Referenz ist für die Ausführung erforderlich. Die in Kapitel 3.6.6 beschriebenen *Scheduler* stellen sicher, dass sich bei der Verarbeitung der in der Nachrichtenschleife befindlichen Elemente zu jedem Zeitpunkt jeweils höchstens ein *Thread* innerhalb eines *Protocols* befindet. Dies schließt das Auftreten von ungewollten Nebenläufigkeitseffekten innerhalb einer *Protocol*-Instanz aus. Es ist darüber hinaus sicherzustellen, dass jeder Aufruf einer Methode oder einer Property des *Protocols* von einem *Thread*, der nicht vom aktuellen *Scheduler* bereit gestellt wird, mit diesem synchronisiert werden muss. Unter der Prämisse, dass ein Entwickler niemals eine andere Funktion als *Start*, *Stop* sowie das Hinzufügen einer Nachricht in die Warteschlange aufruft, ist das Problem der Synchronisierung zu vernachlässigen, da es an den drei angesprochenen Stellen vom Framework implementiert wurde.

Der Aufruf der *Start*-Methode führt zur Initialisierung des *Protocols* und zum Eintritt in die initiale Zustandsmaschine. Die *Start*-Methode setzt zuallererst ein *Lock*, um sicherzustellen, dass die von einem externen *Thread* aufgerufene Methode den *Scheduler*-Thread so lange am Eintritt in das *Protocol* hindert, bis die *Start*-Methode verlassen wurde. Ein *Protocol* kann im Zuge seines Lebenszykluses die fünf verschiedenen Statuszustände *Created*, *Ready*, *Waiting*, *Active* und *Terminated* annehmen.

Der Start eines *Protocols* ist nur aus den Zuständen *Created* und *Terminated* möglich. Sollte sich ein *Protocol* bei Aufruf der *Start*-Methode in einem anderen Zustand befinden, so führt dies unausweichlich zu einer Ausnahme. Sobald das *Protocol* in den geschützten Bereich der *Start*-Methode eingetreten ist, wird die erste Zustandsmaschine mit einem Funktionszeiger auf den *Execute*-Iterator erzeugt und initialisiert. Dieses Verhalten wird von einer Statusänderung des *Protocols* gefolgt. Dieses wird aus einem der Zustände *Created* oder *Terminated* in den Zustand *Waiting* überführt. Die Funktionalität der *Start*-Methode wird mit dem Aufruf der internen *Wakeup*-Methode abgeschlossen.

Die als *internal* markierte *Wakeup*-Methode, die entweder beim Start eines *Protocols* oder beim Hinzufügen einer Nachricht in eine Warteschleife aufgerufen wird, trifft eine Fallunterscheidung in Abhängigkeit vom Status des *Protocols*. Sollte sich die *Protocol*-Instanz im Zustand *Active* befinden, so kann direkt mit der weiteren Nachrichtenverarbeitung fortgefahren werden. Sollte sich das *Protocol* jedoch im Zustand *Waiting* befinden, wie dies beispielsweise direkt nach Aufruf der *Start*-Methode der Fall ist, wird die Instanz zuerst in den Zustand *Ready* versetzt und dann *Wakeup* auf dem ausführenden *Scheduler* aufgerufen. Dieses Verwalten ist deshalb erforderlich, da der vom *Scheduler* bereitgestellte *Thread* aus Effizienzgründen schlafen gelegt wird, sobald keine Nachrichten mehr zu verarbeiten sind. Um dem *Scheduler-Thread* zu signalisieren, dass die Nachrichtenverarbeitung fortgesetzt werden kann, ist dieser durch den Aufruf der *Wakeup*-Methode auf dem *Scheduler*-Objekt aufzuwecken.

Im Anschluss an die Nachrichtenverarbeitung, die im nächsten Abschnitt beschrieben wird, folgt die Terminierung eines *Protocols*. Ein *Protocol* terminiert aus zwei Gründen: erstens, sobald die initiale Zustandsmaschine, der *Execute*-Iterator, ihren Endzustand erreicht hat, oder zweitens durch die Zustellung einer *TerminateMessage*. Die Zustellung einer solchen Nachricht kann durch den Aufruf der *Stop*-Methode eines *Protocols* erreicht werden. Die synchronisierte Methode stellt sicher, ob ein *Protocol* bereits terminiert ist, und stellt andernfalls eine entsprechende Nachricht zu, die die Terminierung aller Zustandsmaschinen und des *Protocols* erzwingt.

Nachrichtenverarbeitung

Die Verarbeitung der in der *broadcastQueue* eines *Protocols* enthaltenen Nachrichten wird vom *Scheduler* angestoßen. Der *Scheduler* ruft dazu die *Run*-Methode des *Protocols* auf. Diese verarbeitet bei jedem Aufruf genau eine Nachricht aus der Warteschlange.

Die Funktionalität der *Run*-Methode, deren Quelltext in Auflistung 3.24 abgedruckt ist, ist in drei Funktionsblöcke aufgeteilt. Der erste Funktionsblock ist für die Entnahme der Nachricht aus der Warteschlange, der zweite für die Zustellung der Nachricht an die Zustandsmaschinen und der dritte für die Zustandsänderung des *Protocols* zuständig.

Die Entnahme einer Nachricht aus der *broadcastQueue* ist an verschiedene Bedingungen geknüpft, die im ersten Funktionsblock der *Run*-Methode in den Zeilen 4 bis 12 implementiert sind. Die Entnahmebedingung ist nur dann erfüllt, wenn sich das *Protocol* in einem der beiden validen Zustände *Ready* beziehungsweise *Created* befindet. Die Ausführung der *Run*-Methode wird andernfalls instantan abgebrochen. Befindet sich das *Protocol* jedoch in einem validen Zustand, so wird eine Nachricht, sofern vorhanden, aus der *broadcastQueue* entnommen und in der Variable *msg* abgelegt. Befindet sich keine Nachricht in der Warteschlange oder handelt es sich um die erste Nachrichtenzustellung an die initiale Zustandsmaschine, was durch die Belegung der Properties *IsFirstRun* mit dem Wert *wahr* gekennzeichnet ist, behält die *msg*-Variable den initialen Wert *null*.

Der zweite Funktionsblock ist für die Zustellung der entnommenen Nachricht an die Zustandsmaschinen zuständig. Hierbei ist das Ergebnis der Zustellung an die initiale Zustandsmaschine, deren Referenz im Klassenfeld *stateMachine* abgelegt ist, von vorrangiger Bedeutung, da das Resultat die Überführung des *Protocols* in einen anderen Zustand bedingen kann. Neben der Zustellung der Nachricht an die initiale Zustandsmaschine, die die Nachrichten an ihre Kinder und somit an alle weiteren Zustandsmaschinen des *Protocols* weiterleitet, wird die Nachricht zudem an alle *DetachedStateMachines* weitergeleitet. Die *DetachedStateMachines* verarbeiten die Nachricht analog zu initialen Zustandsmaschinen in der *Run*-Methode. Das zurückgegebene Resultat ist allerdings nicht relevant für eine mögliche Änderung des *Protocol*-Zustands.

Die Änderung des *Protocol*-Zustands wird ab Zeile 20 implementiert. Die Zustandsänderung ist an das Ergebnis der Nachrichtenzustellung an die initiale Zustandsmaschine geknüpft. Sollte die Maschine durch die Nachrichtenzustellung den Endzustand erreicht haben, terminiert auch die aktuell aktive *Protocol*-Instanz. Die *Protocol*-Terminierung zieht automatisch die Überführung in den Zustand *Terminated* sowie das Löschen aller Elemente aus der *broadcastQueue* nach sich. Terminiert die Maschine nicht im Anschluss an die Nachrichtenverarbeitung, so wird sie in den Zustand *Waiting* überführt. Sollten sich weitere Elemente in der Nachrichtenschlange befinden und zudem das *continueImmediately*-Flag gesetzt sein, endet die Überfüh-

```
1       internal ProtocolStatus Run()
2       {
3           IMessage msg = null;
4           lock (this)
5           {
6               if (this.status != ProtocolStatus.Ready && this.status !=
                    ProtocolStatus.Created)
7                   return this.status;
8               if (this.broadcastQueue.QueueLength > 0 && !this.stateMachine.
                    IsFirstRun && this.status == ProtocolStatus.Ready)
9                   msg = this.broadcastQueue.DequeueMessage();
10
11              this.status = ProtocolStatus.Active;
12          }
13
14          StateMachineStatus sms = this.stateMachine.Run(msg);
15
16          if (this.detachedStateMachines != null && this.detachedStateMachines.
                Count > 0)
17              foreach (AbstractStateMachine sm in this.detachedStateMachines.
                    ToArray())
18                  sm.Run(msg);
19
20          lock (this)
21          {
22              if (sms == StateMachineStatus.Terminated)
23              {
24                  this.broadcastQueue.Clear();
25                  this.status = ProtocolStatus.Terminated;
26                  this.stateMachine = null;
27              }
28              else
29              {
30                  this.status = ProtocolStatus.Waiting;
31                  if (this.broadcastQueue.QueueLength > 0 || this.
                        continueImmediately )
32                  {
33                      this.continueImmediately = false;
34                      this.status = ProtocolStatus.Ready;
35                  }
36              }
37              if (this.parent != null && this.status == ProtocolStatus.Terminated
                    )
38                  this.parent.BroadcastMessage(new ChildTerminatedMessage());
39              return this.status;
40          }
41      }
```

Auflistung 3.24: Implementierung der Run-Methode der ProtocolBase-Klasse

rungsfunktion im Zustand *Ready*, und die Maschine kann instantan mit der weiteren Nachrichtenverarbeitung fortfahren.

Der Funktionsumfang der *Run*-Methode wird mit der Rückgabe des aktuellen *Protocol*-Zustands abgeschlossen. Sollte sich die Maschine jedoch in einem terminalen Endzustand befinden und zudem über ein *Parent-Protocol* verfügen, wird dieses zuvor über die Terminierung der Maschine mittels einer *ChildTerminatedMessage* informiert.

Nachrichtenzustellung

Das Verarbeitungsverfahren der Nachrichten, die sich in der *broadcastQueue* befinden, wurde im vorherigen Abschnitt detailliert beschrieben. Das Verfahren der Nachrichtenzustellung, das gleichbedeutend mit dem Hinzufügen einer Nachricht zur *broadcastQueue* ist, ist Gegenstand des folgenden Abschnitts. Die *ProtocolsBase*-Klasse stellt dazu die drei Methoden *BroadcastMessage*, *BroadcastMessageReliably* und *BroadcastMessageIntern* bereit. Die letztgenannte Methode, die die beiden als *IMessage* beziehungsweise *bool* typisierten Parameter *msg* und *reliable* erwartet, wird von den beiden anderen Varianten der *Broadcast*-Methoden aufgerufen und führt die *Enqueue*-Funktionalität auf der *broadcastQueue* durch. Die beiden Methoden *BroadcastMessage* und *BroadcastMessageReliably* führen zuallererst eine *Lock*-Anweisung aus, um mögliche Effekte der Nebenläufigkeit zu verhindern, und prüfen anschließend den Zustands des *Protocols*. Eine Nachrichtenzustellung erfolgt nur dann, wenn sich das *Protocol* nicht in einem terminierten Zustand befindet und zudem noch keine *TerminateMessage* eingegangen ist. Die Funktionalität der beiden *Broadcast*-Methoden, die die *BroadcastMessageIntern*-Funktion aufrufen, unterscheidet sich lediglich im zweiten Parameter des *BroadcastMessageIntern*-Methodenaufrufs.

Die Semantik der *BroadcastMessage*-Methode, die für die Zustellung aller Nachrichten, die dem *Protocol* extern hinzugefügt werden, verwendet werden soll, besagt, dass der Warteschlange nur solange neue Nachrichten hinzugefügt werden, bis diese die maximal definierte Anzahl an Elementen enthält. Werden weitere Nachrichten hinzugefügt, werden diese verworfen. Die *BroadcastMessage*-Methode informiert den Aufrufer des Rückgabewertes booleanschen Typs über den Erfolg oder Misserfolg des Anfügeprozesses.

Die Semantik der *BroadcastMessageReliably*-Methode ist konträr zur Semantik der *BroadcastMessage*-Methode. Der Aufruf dieser Methode führt unabhängig von der

aktuellen Länge der Warteschlange dazu, dass die Nachricht an die Warteschlange angehangen wird. Dieses Verhalten ist besonders wichtig, um *Protocol*-interne Nachrichten zu verarbeiten, und sollte nicht genutzt werden, um das *Protocol* mit externen Nachrichten zu versorgen, da dieses Verhalten von potentiellen Angreifern genutzt werden könnte, um die Applikation durch Zustellung von Nachrichten in Speichernöte zu bringen.

Receiver Kommandos

Die verschiedenen Ausprägungen der *Receiver*-Klassen sind ausgiebig in den vorangegangenen Abschnitten beschrieben worden. Jede Spezifikation einer Wartebedingung setzt sich aus einer beliebigen Anzahl konkatenierter *Receiver*-Instanzen zusammen, die mittels des *new*-Operators erzeugt werden. Eine beispielhafte Wartebedingung, die den Eingang zweier Nachrichten der Typen *A* und *B* oder einen *Timeout* von *2000* Millisekunden erwartet, müsste wie in Zeile 2 der Auflistung 3.25 dargestellt werden. Das aufgezeigte Instanziierungsverfahren der *Receiver*-Objekte mittels des *new*-Operators hat zwei Nachteile: erstens verlängert sich der resultierende Ausdruck durch die mehrfache Wiederholung des *new*-Schlüsselwortes, und zweitens trägt die Ausdrucksweise nicht zur Erhöhung der Lesbarkeit und des Verständnisses des Quellcodes bei. Die *ProtocolBase* Klasse führt daher statische Methoden ein, hinter denen das Instanziierungsverfahren der *Receiver* verborgen wird. Dies gibt dem Entwickler das Gefühl, ein Kommando der Programmiersprache zu nutzen, und es erhöht zudem enorm die Lesbarkeit und das Verständnis der implementierten Wartebedingung. Eine mittels eines Kommandos erzeugte Wartebedingung, deren Semantik dem aufgezeigten Beispiel folgt, ist in Zeile 5 der Auflistung 3.25 dargestellt und erlaubt somit den direkten Vergleich zur konventionellen Darstellung in Zeile 2.

```
1       //Instance creation syntax
2       yield return new Receiver<A>() & new Receiver<A>() | new TimeoutReceiver
            (2000);
3
4       //Command Syntax
5       yield return Receive<A>() & Receive<A>() | Timeout(2000);
```

Auflistung 3.25: Vergleich der Darstellungsformen von Wartebedingungen

Die *ProtocolBase*-Klasse stellt insgesamt acht Methoden mit verschiedenen Überladungen bereit, um die Kommandosyntax abzubilden. Die im Folgenden vorgestellten statischen Methoden dienen ausschließlich der Vereinfachung des Quellcodes.

Das erste der in alphabetischer Reihenfolge vorgestellten Kommandos ist das *Interval*-Kommando. Es bildet die Instanziierung des *IntervalReceivers* ab und stellt analog zum Konstruktor des *Receivers* zwei Überladungen bereit, die die Übergabe eines *TimeoutHandler* beziehungsweise eines *StateMachineStarter*-Objektes zuzüglich zu den initialen und wiederkehrenden Zeitintervallen erlauben. Der Rückgabetyp dieser statischen *Interval*-Methode ist *IntervalReceiver*.

Der *JoinReceiver* wartet auf die Terminierung einer beliebigen Menge von Zustandsmaschinen, die von der *AbstractStateMachine*-Klasse erben. In der Kommandosyntax wird der *JoinReceiver* als *Join*-Kommando eingeführt. Die *Join*-Methode stellt fünf Überladungen bereit. Diese erlauben die Angabe von einer bis hin zu vier *AbstractStateMachine*-Objekten. Die fünfte Überladung ermöglicht darüber hinaus die Angabe eines beliebig langen *AbstractStateMachine-Arrays*. Die *Join*-Methode liefert einen *JoinReceiver* als Rückgabetyp.

Die Erzeugung von Zustandsmaschinen wurde im Kapitel *StateMachine* ausgiebig beschrieben und in Auflistung 3.23 exemplarisch dargestellt. Neben der Erzeugung von Automaten ist auch das Warten auf ihre Terminierung von größter Bedeutung. Die Wartebedingung wird durch den *StateMachineReceiver* abgebildet. Zur Vereinfachung des Erstellungsverfahren der Zustandsmaschine sowie der Wartebedingung wird das Kommando *Launch* eingeführt. Dieses steht in vier Überladungen bereit und ermöglicht analog zu den vier vorgestellten *StateMachine*-Klassen die Übergabe von keinem bis hin zu drei optionalen Parametern. Das exemplarisch in Auflistung 3.26 dargestellte *Launch*-Kommando verdeutlicht die Vermeidung von Redundanzen, da die immer wiederkehrende Funktionalität der Erzeugung eines *StateMachine*-Objektes, das Setzen der Parameter und die Erzeugung eines *StateMachineReceivers* gekapselt wird. Die *Launch*-Methode gibt den neu erzeugten *StateMachineReceiver* als Rückgabewert an den Aufrufer zurück.

```
1       public static StateMachineReceiver Launch<T1, T2>(StateMachineStarter<T1,
            T2> starter, T1 param1, T2 param2)
2       {
3           StateMachine<T1, T2> sm = new StateMachine<T1, T2>(starter);
4           sm.Parameter1 = para1;
5           sm.Parameter2 = para2;
6           return new StateMachineReceiver(sm);
7       }
```

Auflistung 3.26: Implementierung des Launch-Kommandos mit 2 generischen Parametern

Das vierte Kommando ist für den Start eines als *detached* gekennzeichneten Automaten zuständig und stellt analog zur Erstellung von normalen Zustandsmaschinen

vier Überladungen des *LaunchDetached*-Kommandos bereit. Der Quellcode innerhalb des Methodenrumpfes unterscheidet sich im Vergleich zum *Launch*-Kommando darin, dass die Zustandsmaschine zur Liste der *DetachedStateMachines* des *Protocols* hinzugefügt wird, insofern sie nicht direkt nach ihrer Erstellung terminiert ist. Eine direkte Terminierung könnte beispielsweise aufgrund eines leeren Rumpfes der Zustandsmaschine beziehungsweise aufgrund einer direkten Ausführung einer *yield-break*-Anweisung erfolgen.

Der *ParallelReceiver* wird in Form des *Parallel*-Kommandos abgebildet. Die Aufgabe des vierfach überladenen *Parallel*-Kommandos besteht ausschließlich in der Kapselung des Instanziierungsverfahrens. Das *Parallel*-Kommando gibt somit in jeder Konfiguration eine Instanz des *ParallelReceivers* zurück.

Die Abbildung des *Receiver<T>* erfolgt analog zur Abbildung des *ParallelReceivers*. Das eingeführte *Receive<T>*-Kommando bildet alle drei Konstruktoren des *Receiver<T>* ab. Es erlaubt somit ein bedingungsloses Warten genauso wie die Spezifikation von Filterbedingungen und oder der Angabe eine *ReceiveHandlers*.

Der Zuständigkeitsbereich des vorletzten Kommandos besteht in der Überwachung des Eingangs einer *TerminateMessage*. Das *Terminate*-Kommando kapselt dazu eine *Receiver<TerminateMessage>*-Wartebedingung und ermöglicht dem Entwickler auch an dieser Stelle eine kurze und präzise Ausdruckssyntax.

Der Abschnitt der *Receiver*-Kommandos wird durch die Einführung der *Timeout*-Anweisung abgeschlossen. Dieses kapselt den *TimeoutReceiver*, der einen der am häufigsten verwendeten *Receiver* des Frameworks darstellt. Das *Timeout*-Kommando wird analog zum Konstruktor des *TimeoutReceivers* in zweifacher Überladung angeboten und erlaubt die Konfiguration mit und ohne *TimeoutHandler*.

3.6.6 Scheduler

Die Zuweisung von Rechenzeit und CPU-Zyklen an die *Protocol*-Instanzen erfolgt über *Scheduler*, die als Teil des Gears4Net Frameworks auf *User-Mode* implementiert wurden und den letzten Abschnitt des Implementierungskapitels bilden. Jedem *Protocol* wird genau ein dedizierter *Scheduler* zugewiesen. Eine *Scheduler*-Instanz kann hingegen beliebig viele *Protocol*-Objekte verwalten. Zudem ist es möglich, mehrere *Scheduler* innerhalb eines Prozesses zu erzeugen und parallel auszuführen.

Die Aufgabe des *Schedulers* besteht in der Zuweisung der Rechenzeit an die ihm zugewiesenen aktiven und rechenbereiten *Protocol*-Instanzen. Die Zuweisung der Rechen-

zeit erfolgt durch den Aufruf der *Run*-Methode des ausgewählten *Protocols*. Verfügt ein *Scheduler* zu einem beliebigen Zeitpunkt über kein rechenbereites *Protocol*, so legt der *Scheduler* seinen für die Ausführung zuständigen *Thread* schlafen, bis eines der verwalteten *Protocol*-Objekte in einen rechenbereiten Zustand überführt wird. Ein *Protocol* ist genau dann rechenbereit, wenn dieses über eine Nachricht in seiner Warteschlange verfügt. Das Einstellen einer Nachricht in die Warteschleife erfolgt in eine einem Kontext eines externen *Threads*. Somit kann bei jedem Einstellen einer Nachricht sichergestellt werden, dass ein potentiell schlafender *Scheduler-Thread* aufgeweckt wird.

Das verwendete *Scheduling*-Verfahrens ist abhängig vom Einsatzzweck des oder der *Protocol*-Objekte, die von einer *Scheduler*-Instanz ausgeführt werden. Das Gears4Net Framework stellt die zwei unterschiedlichen *Scheduling*-Konzepte *STAScheduler* und *WinFormsScheduler* vor, die ihre gemeinsam genutzte Funktionalität in der abstrakten Basisklasse *Scheduler* ablegen. Die Entwicklung weiterer *Scheduler* kann jederzeit durch die Erstellung einer von *Scheduler* erbenden Klasse erreicht werden.

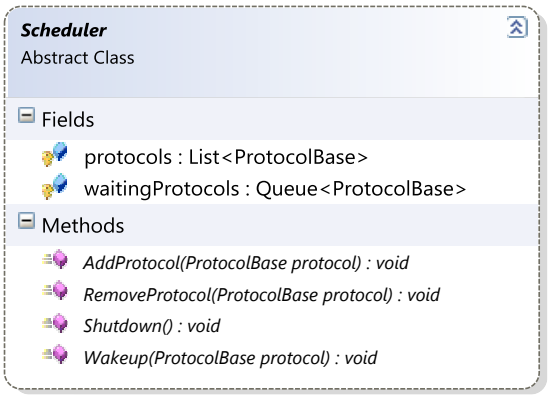

Abbildung 3.16: Klassendiagramm der Scheduler-Klasse

Die abstrakte *Scheduler*-Klasse, deren Klassendiagramm in Abbildung 3.16 dargestellt ist, verfügt über vier abstrakte Methoden, die von jedem *Scheduler* implementiert werden müssen. Das Methodentupel *AddProtocol*- und *RemoveProtcol* fügt ein *Protocol* hinzu beziehungsweise entfernt dieses aus der internen Liste des *Schedulers* und führt unter Umständen die dritte abstrakte Methode, die *Shutdown*-Funktion, durch, falls dem *Scheduler* kein *Protocol* mehr zugewiesen ist. Die vierte und letzte abstrakte Methode der *Scheduler*-Klasse ist die *Wakeup*-Methode. Diese überführt den gegebenenfalls schlafenden *Scheduler-Thread* in einen aktiven Zustand und fügt das *Protocol*, das der *Wakeup*-Methode übergeben wurde, in die Liste der aktiven *Protocol*-Instanzen hinzu.

3.6.6.1 STAScheduler

Der meist verwandte *Scheduler* ist der *STAScheduler*, der mit vollem Namen als *Single Thread Appartment Scheduler* bezeichnet wird. Die Abgrenzung der *Threading*-Modelle *STA* von *MTA* wird in [91] vorgestellt und an dieser Stelle nicht näher erläutert. Innerhalb des Konstruktors der *STAScheduler*-Klasse wird ein neuer *Thread* erzeugt, in dessen Kontext die *Protocol*-Instanzen des *Scheduler* ausgeführt werden. Der neue *Thread* wird im *STA*-Modus ausgeführt, und ihm wird optional zu *Debugging*-Zwecken ein dedizierter Name zugewiesen, der über den Klassenkonstruktor angegeben werden kann. Der *Thread* erhält zudem einen Funktionszeiger auf die Methoden, der bei Start des *Threads* ausgeführt werden soll. Im vorgestellten Fall ist dies ein Zeiger auf die in Auflistung 3.27 dargestellte private *DoScheduling*-Methode, die die Funktionalität für das eigentliche *Scheduling*-Verfahren beinhaltet.

Die *DoScheduling*-Methode besteht aus zwei ineinander verschachtelten *while*-Scheifen, deren Terminierungsbedingungen niemals erfüllt werden können. Die innere Schleife wird solange durchlaufen, bis kein *Protocol* mehr in der Liste der wartenden *Procotol*-Objekte vorhanden ist. Dies ist gleichbedeutend mit der Tatsache, dass keine *Protocol*-Instanz rechenbereit ist und somit keine Rechenzeit zugeteilt werden kann. Sobald dies der Fall ist, springt der Quellcode mittels einer *break*-Anweisung aus der inneren Schleife heraus. Der ausführende *Thread* wird in der umgebenden Schleife solange schlafen gelegt, bis ein rechenbereites *Protocol* zur Verfügung steht.

```
1       private void DoScheduling()
2       {
3           while (true)
4           {
5               this.wakeup.WaitOne();
6
7               while (true)
8               {
9                   if (this.shutdown)
10                      return;
11
12                  ProtocolBase protocol = null;
13                  lock (this)
14                  {
15                      if (this.waitingProtocols.Count == 0)
16                          break;
17                      protocol = this.waitingProtocols.Dequeue();
18                  }
19
20                  ProtocolStatus status = protocol.Run();
21
22                  lock (this)
```

```
23                  {
24                      switch (status)
25                      {
26                          case ProtocolStatus.Created:
27                              break;
28                          case ProtocolStatus.Ready:
29                              this.waitingProtocols.Enqueue(protocol);
30                              break;
31                          case ProtocolStatus.Waiting:
32                              break;
33                          case ProtocolStatus.Terminated:
34                              this.RemoveProtocol(protocol);
35                              break;
36                      }
37                  }
38              }
39          }
40      }
```

Auflistung 3.27: Schedulingverfahren des STASchedulers

Ein Blick auf den Quellcode der *DoScheduling*-Methode in Auflistung 3.27 sowie der *Wakeup*-Methode in Auflistung 3.28 verdeutlicht dieses Verfahren. Sobald die äußere *while*-Schleife betreten wird, wird die *WaitOne*-Methode des *AutoResetEvent*-Klassenfeldes *wakeup* aufgerufen. Dieses blockiert den aufrufenden *Thread* solange, bis von einem anderen *Thread* die Methode *Set* auf demselben Klassenfeld ausgeführt wird. Der Aufruf der *Set*-Methode erfolgt in der *Wakeup*-Funktion des *Schedulers*. *Wakeup* wird immer dann aufgerufen, wenn ein neues *Protocol* erstellt wurde oder der Nachrichtenwarteschlange eines *Protocols* eine Nachricht hinzugefügt wurde. Die *Wakeup*-Methode fügt bei ihrem Aufruf das übergebene *Protocol* zur Liste der rechenbereiten *Protocol*-Objekte hinzu und ruft im Anschluss die *Set*-Methode des *AutoResetEvents*. Dieser Aufruf hebt die Blockierung des *Scheduler-Threads* auf und erlaubt den Eintritt in die innere *while*-Schleife in Zeile 7 der*DoScheduling*-Methode.

```
1       public override void Wakeup(ProtocolBase protocol)
2       {
3           lock (this)
4           {
5               if (!this.waitingProtocols.Contains(protocol))
6                   this.waitingProtocols.Enqueue(protocol);
7               this.wakeup.Set();
8           }
9       }
```

Auflistung 3.28: Wakeup-Methode des STASchedulers

Der innere Schleifenrumpf beginnt mit der Prüfung der Terminierungsbedingung der *Scheduler*-Instanz und springt bei Erfüllung dieser aus der *DoScheduling*-Methode heraus. Andernfalls wird mit der Entnahme eines *Protocols* aus der *waitingProtocols*-Liste fortgefahren. Sollte diese keine Elemente enthalten, erfolgt ein Sprung aus der inneren *while*-Schleife. Dies führt automatisch zu einer Blockierung des aktuellen *Threads*, bis ein neues *Protocol* über den Aufruf der *Wakeup*-Methode in die *waitingProtocols*-Liste aufgenommen wird. Andernfalls wird auf dem entnommenen *Protocol* die *Run*-Methode aufgerufen, die die Nachrichtenverarbeitung innerhalb des *Protocols* anstößt. Das Resultat des *Run*-Methodenaufrufs wird ab Zeile 24 in Auflistung 3.27 differenziert betrachtet. Sollte ein *Protocol* den Endzustand erreicht haben, wird dieses automatisch aus der *Protocol*-Liste des *Schedulers* entfernt. Ist ein *Protocol* jedoch im Zustand *Ready*, wird dieses direkt wieder in die *waitingProtocols*-Liste aufgenommen, um beim nächsten Schleifendurchlauf abermals aufgerufen zu werden.

Das Schedulingverfahren des *STAScheduler* ermöglicht somit eine effiziente Zuweisung der Rechenzeit an eine beliebige Anzahl an *Protocol*-Instanzen und ermöglicht zugleich einen schonenden Umgang mit den Ressourcen des Rechners, da sich ein nicht verwendeter *Scheduler-Thread* schlafen legen kann und keine unnötige CPU-Zyklen verschwendet.

3.6.6.2 WinFormsScheduler

Unabhängig von der zugrundeliegenden Plattform, basiert nahezu jede moderne *Rich-Client*-Applikation mit aufwändigem graphischen Benutzerinterface auf einer *Multithreading*-Architektur, um langlaufende Berechnungen, Analysen und Dienstanfragen von der Benutzeroberfläche zu trennen. Neben den bereits angesprochenen üblichen Synchronisierungsproblemen erfordern Anwendungen mit einer graphischen Benutzeroberfläche einen zusätzlichen Synchronisationsschritt, sobald auf einzelne Steuerelemente zugegriffen werden soll. Die graphischen Steuerelemente aller großen Frameworks sind nicht *Thread*-sicher, da bei deren Entwicklung die Darstellungsgeschwindigkeit höchste Priorität hat und eine permanente Synchronisierung diese erheblich reduzieren würde. Dieses resultiert in der Verlagerung des Synchronisierungsprozesses auf den Applikationsentwickler. Die in Kapitel 2.5 aufgezeigten Lösungsvorschläge müssen somit aufwändig implementiert werden.

Das Konzept des *WinFormsScheduler* ist nun darauf ausgelegt, die laufenden *Protocol*-Instanzen einer Gears4Net Applikation im *GUI-Thread* einer *Windows-Forms*-Applikation auszuführen. Der Vorteil dieses Verfahrens bei Applikationen mit gra-

phischer Benutzeroberfläche ist offenkundig, da jede Verarbeitung einer Nachricht aus der *Protocol*-Warteschlange im Kontext des *Scheduler-Threads* ausgeführt wird, der dem *GUI-Thread* der Applikation entspricht. Der Zugriff auf die Steuerelemente der Oberfläche erfolgen nun automatisch im richtigen *Thread* und machen die beschriebenen Synchronisierungsverfahren obsolet.

Bei der Implementierung eines *Schedulers* mit der Eigenschaft, dass *Protocol*-Instanzen und *GUI* von ein und demselben *Scheduler-Thread* ausgeführt werden, entsteht ein Interessenkonflikt. Ansprechbarkeit und Reaktionsvermögen sind die höchsten Güter bei der Entwicklung von graphischen Benutzeroberflächen. Dies steht im Kontrast zu der bisherigen *Scheduler*-Implementierung. Diese blockiert den ausführenden *Scheduler-Thread*, insofern keine Nachrichten vorhanden sind, die in einem *Protocol* verarbeitet werden können. Ein solches Verhalten ist jedoch nicht tragbar, wenn die *GUI* im gleichen *Thread* ausgeführt wird, da dieses ein sofortiges Einfrieren der Benutzeroberfläche zur Folge hätte. Der alternative Einsatz von *polling*-Mechanismen beziehungsweise *Busy-Waiting*-Strategien würde zur Verschwendung von CPU-Zyklen führen und die Reaktionsfähigkeit der Oberfläche einschränken.

Implementierungsidee

Die Implementierungsidee des *WinFormsScheduler* basiert daher auf der Verwendung der Windows-Nachrichtenschleife, die für die Zustellung von Nachrichten an alle graphischen Steuerelemente auf dem Windows-Desktop verantwortlich ist. Das in [95] vorgestellte Konzept erlaubt das asynchrone Senden und Empfangen von Nachrichten aus der Windows-Warteschleife. Die der *Scheduler*-Implementierung zugrunde liegende Idee besagt, dass ein *WinFormsScheduler* analog zum *STAScheduler* in der *DoScheduling*-Methode die aktiven *Protocol*-Instanzen aufruft und die Nachrichtenverarbeitung anstößt. Im Gegensatz zum *STAScheduler* blockiert die *DoScheduling*-Methode jedoch nicht, sondern läuft zu Ende, wenn keine Verarbeitungsschritte mehr durchgeführt werden, und gibt damit die Rechenzeit für die graphische Oberfläche frei. Die Information, wann mit der weiteren Verarbeitung neuer Nachrichten fortgefahren werden kann, wird über die Windows-Nachrichtenschleife zugestellt. Sobald der *broadcastQueue* eines *Protocols* eine Nachricht zugestellt wird, erzeugt diese eine *WndMessage*, adressiert diese an die eigene Oberfläche und legt sie anschließend in die Windows-Nachrichtenschleife. Sobald die Nachricht die eigene Applikation erreicht, kann diese verarbeitet und an den *Scheduler* weitergegeben werden. Der *Scheduler* ruft daraufhin die *DoScheduling*-Methode auf und fährt mit der Verarbeitung fort, ohne dass eine Blockade des *Scheduler-Threads* notwendig gewesen ist.

Für die Implementierung des vorgestellten Konzeptes sind zwei elementare Bausteine zu konzipieren. Der erste Baustein behandelt die asynchronen Kommunikationsmechanismen der Windows-Nachrichtenschleife, der zweite die Implementierung des *Schedulers* in Anbetracht der definierten Eigenschaften.

Die Umsetzung des ersten Bausteins erfolgt in der in Auflistung 3.29 dargestellten *WinFormsSchedulerHelperControl*-Klasse. Die Windows-Nachrichtenschleife stellt ihre Nachrichten ausschließlich an registrierte Steuerelemente zu. Aus diesem Grund erbt die *WinFormsSchedulerHelperControl*-Klasse von der Basisklasse aller Steuerelemente, der Klasse *System.Windows.Forms.Control*. Die *WinFormsSchedulerHelperControl*-Klasse verfügt über einen Konstruktor, zwei Methoden und eine externe Methodensignatur. Der Konstruktor, dem die Referenz auf ein *WinFormsScheduler*-Objekt übergeben wird, legt dieses in der Klassenvariable *scheduler* ab und fährt mit der Erzeugung des Steuerelementes fort. Der Aufruf der *Hide*-Methode setzt die Sichtbarkeitseigenschaft des Steuerlementes auf *falsch* und sorgt damit dafür, dass das registrierte Element niemals für den Benutzer sichtbar sein wird. Der beschriebene Methodenaufruf wird durch den Funktionsaufruf *CreateControl* in Zeile 19 gefolgt. Der etwas missverständliche Methodenname führt nicht zur Erzeugung des Steuerelementes, sondern forciert lediglich die Erstellung eines *Window-Handels*, das für die spätere Adressierung der Nachrichten notwendig ist.

Die beiden Methoden *WndProc* und *SendMessage* sind für den Empfang und das Versenden der Nachrichten zuständig. Die mit dem Schlüsselwort *override* gekennzeichnet überschriebene Methode *WndProc* in Zeile 24 verarbeitet alle eingehenden Nachrichten, die an dieses Steuerelement adressiert wurden. Die Adresse eines Steuerelementes ist durch sein *Window-Handle* bestimmt, das im vorliegenden Fall im Konstruktor der Klasse erzeugt wurde. Die *Wnd-Proc*-Methode überprüft jede eingehende Methode, ob diese einem speziellen Typ entspricht. Ist dies der Fall, wird die *DoScheduling*-Methode des *WinFormsScheduler* aufgerufen und mit dem *Scheduling* begonnen. Im Anschluss an die Prüfung werden alle Nachrichten an die Basisimplementierung der *Control*-Klasse weitergeleitet.

```
1   internal class WinFormsSchedulerHelperControl : Control
2   {
3       private const uint WM_APP = 0x8000;
4       private WinFormsScheduler scheduler;
5       private int hwnd = 0;
6
7       [System.Runtime.InteropServices.DllImport("user32.dll")]
8       public static extern int PostMessage(
9             int hWnd,      // handle to destination window
10            uint Msg,      // message
11            long wParam,   // first message parameter
12            long lParam    // second message parameter
13      );
14
15      public WinFormsSchedulerHelperControl(WinFormsScheduler scheduler)
16      {
17          this.scheduler = scheduler;
18          this.Hide();
19          this.CreateControl();
20          this.hwnd = this.Handle.ToInt32();
21      }
22
23      protected override void WndProc(ref Message m)
24      {
25          if (m.Msg == WM_APP)
26              this.scheduler.DoScheduling();
27          base.WndProc(ref m);
28      }
29
30      public void SendMessage()
31      {
32          PostMessage(this.hwnd, WM_APP, 0, 0);
33      }
34  }
```

Auflistung 3.29: Implementierung der WinFormsSchedulerHelperControl-Klasse

Das Versenden einer Nachricht erfolgt über die *SendMessage*-Methode, die eine an sich selbst adressierte Nachricht in die Windows-Nachrichtenschleife einstellt und diese über die Konstante *WM_APP* als *Scheduling*-Nachricht spezifiziert. Die Klassenbibliothek des Microsoft .NET Frameworks stellt keine Funktionalität für einen schreibenden Zugriff auf die Windows-Nachrichtenschleife bereit. Aus diesem Grund ist es nötig, die *postMessage*-Methode aus der *user32.dll* zu importieren. Die Signatur der Methode ist in den Zeilen 7 bis 13 dargestellt. Auf eine als extern deklarierte Methode kann direkt aus dem verwalteten C#-Quelltext zugegriffen werden, wie es beim Aufruf der *PostMessage*-Methode in Zeile 32 aufgezeigt wurde.

Die *WinFormsSchedulerHelperControl*-Klasse ist somit zum Empfangen und Versenden von Nachrichten aus der Windows-Warteschlange bereit. Ausstehend ist lediglich die Registrierung der Objektinstanz durch den Aufruf der Methode *Application.Run*,

die während der Initialisierung der im Folgenden vorgestellten *WinFormsScheduler*-Klasse ausgeführt wird.

Die *WinFormsScheduler*-Klasse leitet analog zur *STAScheduler*-Klasse von der gemeinsamen abstrakten Basisklasse *Scheduler* ab und implementiert die darin vorgesehenen Methoden *AddProtocol*, *RemoveProtocol*, *Wakeup* und *Shutdown*, welche aufgrund der Ähnlichkeit zur *STAScheduler*-Implementierung im Folgenden nicht betrachtet werden. Mit Ausnahme der *DoScheduling*-Methode hebt sich die *WinFormsScheduler*-Implementierung an zwei weiteren Punkten von der Darstellung des vorherigen *Schedulers* ab. Der *WinFormsScheduler* erzeugt in seinem Konstruktor einen eigenen *Thread* und erklärt diesen durch den Aufruf der *Application.Run*-Methode zum aktuellen *GUI-Thread*, was die Registrierung einer *WinFormsSchedulerHelperControl*-Instanz und die damit verbundenen Kommunikationsmöglichkeiten erlaubt. Die zweite Unterscheidung offenbart sich in der Implementierung der *WakeupInternal*-Methode, die dem *Scheduler* signalisiert, dass weitere *Protocol*-Instanzen zur Ausführung bereitstehen. Da das vorgestellte Konzept keine Blockierung des aktuellen *Scheduler-Threads* kennt und diese somit nicht aufheben kann, ist es Aufgabe der *WakeupInternal*-Methode, durch das Einstellen einer Nachricht in die Windows-Nachrichtenschleife die Möglichkeit der weiteren Verarbeitung zu signalisieren.

Der Algorithmus des *Scheduling*, der in der *DoScheduling*-Methode implementiert wurde, ähnelt dem des *STASchedulers*. Dieser beinhaltet jedoch keine ineinander verschachtelten niemals terminierenden *while*-Schleifen, sondern basiert auf dem in Auflistung 3.30 abgedruckten Quelltext. Der Eintritt in die *DoScheduling*-Methode beginnt mit der Prüfung der Terminierungsbedingung des *Schedulers*. Sollte diese erfüllt sein, wird die Ausführung des *Schedulers* unterbrochen. Der Aufruf der *ShutdownRequired*-Methode in Zeile 6 und später in Zeile 48 zieht neben der Prüfung der Terminierungsbedingung auch gleichzeitig den Aufruf der *Application.ExitThread*-Funktion nach sich, insofern die Terminierungsbedingung erfüllt sein sollte.

Andernfalls wird mit der Nachrichtenzustellung innerhalb der *for*-Schleife in den Zeilen 11 bis 44 begonnen. Der *Scheduler* wählt dazu das nächste *Protocol* aus der Liste der aktiven *Protocol*-Instanzen. Sollte keine Instanz in der Liste verfügbar sein, verlässt der *Scheduler* die *for*-Schleife und setzt die Ausführung mit dem folgenden Quellcode fort. Die explizite Nachrichtenzustellung erfolgt in der inneren der beiden *for*-Schleifen durch den Aufruf der *Run*-Methode des gewählten *Protocols* ab Zeile 22. Jede *Protocol*-Instanz wird dabei mindestens einmal und maximal so häufig aufgerufen, wie Nachrichten in der Warteschlange verfügbar sind. Es ist dabei zu

```
1       internal void DoScheduling()
2       {
3           int count = 0;
4           lock (this)
5           {
6               if (ShutdownRequired())
7                   return;
8               count = this.waitingProtocols.Count;
9           }
10
11          for (int i = 0; i < count; ++i)
12          {
13              ProtocolBase protocol = null;
14              lock (this)
15              {
16                  if (this.waitingProtocols.Count == 0)
17                      break;
18                  protocol = this.waitingProtocols.Dequeue();
19              }
20
21              int qlength = Math.Max(1, protocol.QueueLength);
22              for (int k = 0; k < qlength; ++k)
23              {
24                  ProtocolStatus status = protocol.Run();
25                  if (status == ProtocolStatus.Ready)
26                  {
27                      if (k == qlength - 1)
28                      {
29                          lock (this)
30                          {
31                              this.waitingProtocols.Enqueue(protocol);
32                          }
33                      }
34                      continue;
35                  }
36                  if (status == ProtocolStatus.Waiting)
37                      break;
38                  if (status == ProtocolStatus.Terminated)
39                  {
40                      this.RemoveProtocol(protocol);
41                      break;
42                  }
43              }
44          }
45
46          lock (this)
47          {
48              if (ShutdownRequired())
49                  return;
50              this.waiting = false;
51              if (this.waitingProtocols.Count > 0)
52                  this.WakeupInternal();
53          }
54      }
```

Auflistung 3.30: Schedulingverfahren des WinFormsScheduler

beachten, dass die *Run*-Methode einer *Protocol*-Instanz auch dann aufgerufen wird, wenn die Warteschlange der Instanz kein Element enthält. Dieser Aufruf ist beispielsweise für die Initialisierung einer neuen Zustandsmaschine nötig. Das weitere Verfahren ist abhängig vom Zustand, den das *Protocol*-Objekt nach Aufruf der *Run*-Methode eingenommen hat. Sollte sich das *Protocol* im Zustand *Ready* befinden, so wird es direkt wieder in die Liste der wartenden *Protocol*-Instanzen aufgenommen. Im Zustand *Terminated* wird die Instanz aus der Liste der zu betreuenden Objekte des *Schedulers* entfernt.

Die *DoScheduling*-Methode wird mit einer weiteren Prüfung der Terminierungsbedingung abgeschlossen. Sollten sich weitere *Protocol*-Objekte in der Warteschlange befinden, wird dies durch den Aufruf der *WakeupInternal*-Methode angezeigt und führt zu einer wiederholten Ausführung der *DoScheduling*-Methode.

Der *WinFormsScheduler* findet seine Anwendung in der in Kapitel 5.4 vorgestellten Simulationsapplikation *Peers@Play-Inspector* und wird in diesem Zusammenhang ausgiebig vorgestellt.

3.7 Anwendungsszenarien und Beispiele

Das vorangegangene Kapitel hat die Implementierung der *Gears4Net*-Architektur in allen Einzelheiten beschrieben und ausgewählte Detaillösungen mit Quellcodebeispielen hinterlegt. Der folgende Abschnitt erörtert drei alltägliche Problemstellungen der Entwicklung paralleler und verteilter Systeme und zeigt ihre Umsetzung auf Basis der Implementierung des *Gears4Net*-Modells.

3.7.1 Grundlagen der Netzwerkprogrammierung

Ein typisches und abermals implementiertes Szenario der Netzwerkprogrammierung ist der Aufbau einer *TCP*-Verbindung und der anschließende Austausch von Daten gemäß der definierten Anwendungslogik. Begleitet wird das beschriebene Verfahren von *Timeout*-Mechanismen, die festlegen, innerhalb welches Zeitintervalls eine Verbindung aufgebaut sein muss und innerhalb welches Intervalls die Verarbeitung der Applikationslogik inklusive des Nachrichtentransfers stattzufinden hat.

Das beschriebene Szenario ist in Form eines Automaten in Abbildung 3.17 dargestellt. Die Initialisierung und der Aufbau einer Netzwerkverbindung zu einer entfernten Gegenstelle beginnt ausgehend von dem am oberen Rand des Automaten darge-

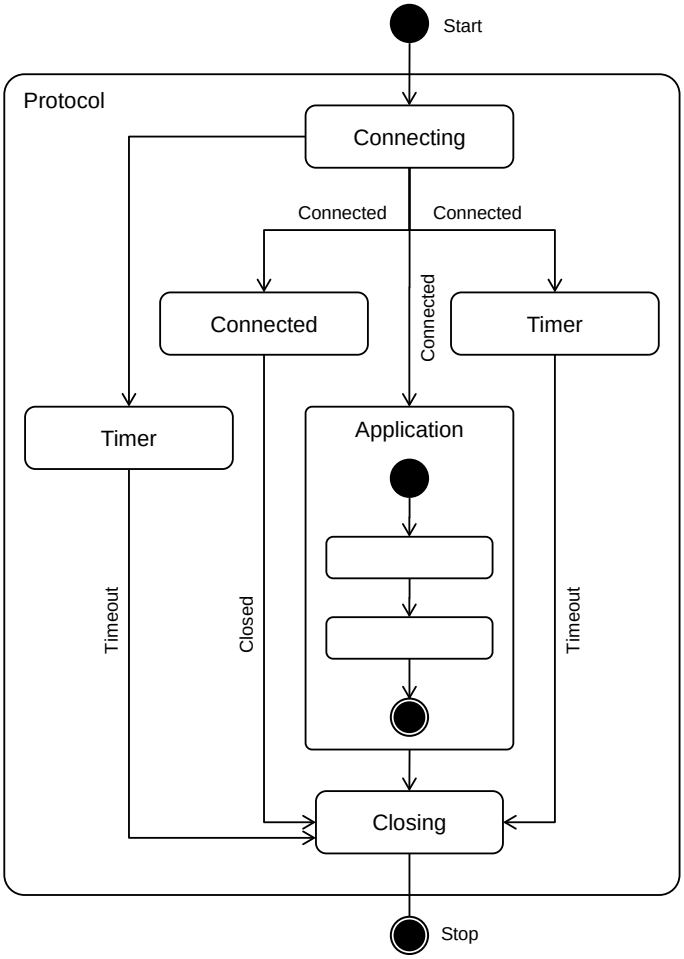

Abbildung 3.17: Zustandsautomat zum Aufbau einer Netzwerkverbindung und Ausführung der Applikationslogik

stellten Startzustands. Der gesamte Automat einschließlich des Verbindungsaufbaus und der Anwendungslogik muss innerhalb eines definierten Zeitintervall abgearbeitet werden. Ist dies nicht der Fall, terminiert der Automat. Das Zeitintervall zur Automatenterminierung wird durch den *Timer* an der linken Seite des Diagramms definiert. Sobald die Verbindung zur Gegenstelle hergestellt ist, dritt die Zustandsmaschine in drei parallele Zustände ein. Der Zustand *Connected* wird solange gehalten, bis die Verbindung unterbrochen beziehungsweise geschlossen wird. Der zweite Zustand wird durch den *Timer* auf der linken Seite des Diagramms definiert. Dieser prüft, ob die Ausführung der Anwendungslogik ein festgelegtes Zeitintervall nicht überschreitet. Der dritte und letzte der parallelen Zustände ist der Zustand *Application*. Dieser führt die Anwendungslogik aus und beinhaltet gegebenenfalls weitere interne Zustände. Sobald eine Änderung in einem der vier Zustände vorgenommen

wird, terminiert das aufgezeigte *Protocol* inklusive aller verschachtelten Subautomaten.

Das auf den ersten Blick simple Szenario zeigt seine vollständige Komplexität im dargestellten Automatendiagramm und der Tatsache, dass in einer realen Anwendung eine große Menge an simultanen Netzwerkverbindungen aufgebaut werden müssen, wie dieses beispielsweise bei einem klassischen *Web-Server* der Fall ist. Zudem bietet dieses Szenario durch die Vielzahl an parallelen *Threads*, beispielsweise durch die asynchron ausgeführten *Timer*-Objekte, einen nahezu perfekten Nährboden für *Race-Conditions*.

Die im folgenden aufgezeigte Implementierung des vorgestellten Automaten auf Basis des *Gears4Net*-Programmiermodells verdeutlicht daher abermals die kompakte Darstellungsform sowie die Fähigkeit des Modells, ohne jegliche Synchronisierungsmechanismen auszukommen.

```
1       public IEnumerator<ReceiverBase> Execute()
2       {
3           OpenConnectionAsync();
4           yield return (Receive<ConnectedMsg>() + Launch(StartApplication
5                       | Timeout(5000))
6                       | Receive<ConnectionClosedMsg>())
7                       | Timeout(10000);
8           CloseConnection();
9       }
```

Auflistung 3.31: Gears4Net Implementierung zum Aufbau einer Netzwerkverbindung und Ausführung der Applikationslogik

Der in Auflistung 3.31 dargestellte Quelltext beginnt im Anschluss an die Deklaration des Iterators mit der Initialisierung und dem Aufbau der Netzwerkverbindung durch den Aufruf der Methode *OpenConnectionAsync* und tritt im Anschluss in die Implementierung der *yield*-Wartebedinung ab Zeile 3 ein. Die Wartebedingung besteht aus zwei Subbedingungen. Der erste Bestandteil der gesamten Wartebedingung ist in den Zeilen 3 bis 5 darstellt und wird durch die umschließenden runden Klammern charakterisiert. Die zweite Subbedingung besteht ausschließlich aus dem *Timeout*-Receiver, der das Intervall von *10000 ms* in Zeile 6 festlegt. Die zuletzt genannte Subbedingung entspricht dem äußeren, an der linken Seite des Automatendiagramms dargestellten *Timer*-Zustand. Die erste Subbedingung implementiert die drei parallelen Zustände, in die bei erfolgreichem Verbindungsaufbau eingetreten wird. Sobald eine *ConnectedMsg* eintrifft, wird die Methode *StartApplication* aufgerufen, die die Anwendungslogik ausführt. Parallel dazu wird ein weiterer *Timeout*-

Receiver mit einem Zeitintervall von *5000 ms* erzeugt, innerhalb dessen die Applikationslogik ausgeführt sein muss. Dieser *Receiver* entspricht dem rechten *Timer* im Zustandsdiagramm. Der letzte Teil der ersten Subbedingung wird durch den *Receiver* in Zeile 5 gebildet, der auf den Eingang einer *ConnectionClosedMsg* wartet, die eintrifft, falls die Verbindung zur Gegenstelle abreißt oder geschlossen wird.

Sobald die gesamte Wartebedingung erfüllt ist, wird die geöffnete Netzwerkverbindung geschlossen und der Automat terminiert. Das *Gears4Net*-Modell ermöglicht zudem die Ausführung von beliebig vielen parallelen *Protocol*-Instanzen. Somit kann für den Aufbau jeder Netzwerkverbindung ein eigener Automat instantiiert werden, der auf einer beliebigen CPU des Rechners parallel zu den anderen Instanzen ausgeführt wird.

3.7.2 Drei-Wege-Handshake

Der Drei-Wege-*Handshake* [23, 26] ist ein klassisches Verfahren zum Aufbau der Kommunikation zwischen zwei Endpunkten, das beispielsweise für den Aufbau einer *TCP*-Verbindung eingesetzt wird. Um eine Verbindung zwischen den Endpunkten a und b herzustellen, sendet Endpunkt a eine *SYN*-Nachricht an den Endpunkt b. Dieser antwortet mit einer *SYN-ACK*-Nachricht, die anschließend von Endpunkt a mit einer *ACK*-Nachricht bestätigt wird. Die beiden Endpunkte a und b legen zudem beim Versenden der *SYN*- und *SYN-ACK*-Nachricht ein Zeitintervall fest, innerhalb dessen die korrespondierende Antwort eintreffen muss, da andernfalls der *Handshake* als gescheitert betrachtet werden muss.

Das vorgestellte Drei-Wege-*Handshake*-Protokoll verfügt über zwei unterschiedliche Rollen, die durch die Endpunkte a und b symbolisiert werden und im Folgenden als *Initiator* und *Empfänger* benannt werden. Die erste Rolle, der *Initiator*, verschickt eine *SYN*-Nachricht an einen *Empfänger* und wartet auf den Eingang der Antwortnachricht des Typs *SYN-ACK*, um im Anschluss die bestätigende *ACK*-Nachricht zu versenden. Das Verhalten der zweiten Rolle, dem *Empfänger*, verläuft invers. Dieser wartet auf den Eingang einer *SYN*-Nachricht, um mit einer *SYN-ACK*-Nachricht zu antworten und anschließend auf die Bestätigung in Form einer *ACK*-Nachricht zu warten. Das beschriebene Verfahren ist sowohl für den Empfänger, als auch für den Initiator im Zustandsdiagramm in Abbildung 3.18 verzeichnet, und bildet die Grundlage für die im Folgenden beschriebene Implementierung.

Die Implementierung des Drei-Wege-*Handshake*-Protokolls auf Basis des *Gears4Net*-Programmiermodells ist in Auflistung 3.32 auszugsweise dargestellt. Auf die Metho-

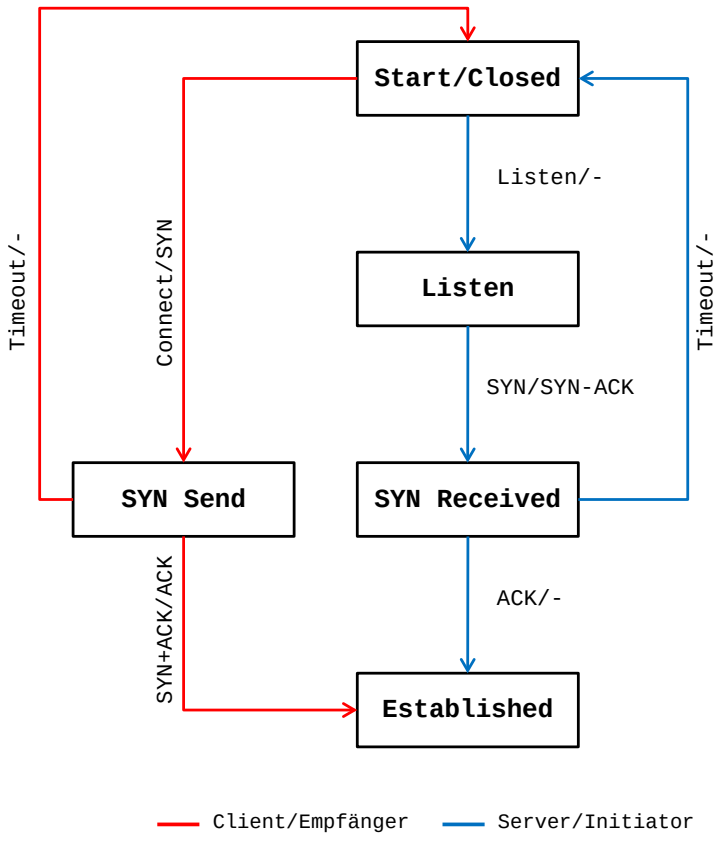

Abbildung 3.18: Zustandsautomat des Drei-Wege-Handshake (Client und Server)

den zum Empfangen und Versenden der drei Nachrichtentypen ist verzichtet worden, da diese keinen Beitrag zum Verständnis der Protokollimplementierung leisten. Die dargestellte Klasse *HandshakeProtocol* erbt wie jede *Gears4Net-Protocol*-Instanz von der Basisklasse *ProtocolBase* und stellt somit die initiale Zustandsmaschine *Execute* bereit, die beim Start der *Protocol*-Instanz aufgerufen wird. Die Klasse verfügt zudem über das Signal *exitSignal* und einen Konstruktor, dem der der Instanz zugewiesene *Scheduler* übergeben wird. Die Methoden *Execute* und *HandleIncomingHandshakeRequest* implementieren die *Empfängerrolle*, die Methoden *InitiateHandshake* und *InitiateHandshakeRequest* die Rolle des Initiators.

Die *yield-return*-Anweisung in Zeile 13 des Quellcodebeispiels enthält neben der Wartebedingung für *SynMsg*-Nachrichten die durch das *exitSignal* bereitgestellte Terminierungsbedingung. Sobald die *Emit*-Methode des Signals aufgerufen wird, terminiert die vollständige *HandshakeProtocol*-Instanz. Der vordere Teil der *yield-return*-Anweisung, das *Parallel*-Kommando, erzeugt zusammen mit der umschlossenen *Receive<SynMsg>*-Anweisung eine Wartebedingung, die jede eingehende *SynMsg*-Nachricht annimmt und diese an eine neu erzeugte Untermaschine weiterleitet. Diese

trägt den Namen *HandleIncomingHandshakeRequest* und ist für die weitere Verarbeitung des Protokollablaufes verantwortlich.

Dem Drei-Wege-*Handshake*-Protokoll folgend versendet die *SendSynAck* in Zeile 18 eine *SynAckMsg*-Nachricht an den Initiator des *Handshakes* und wartet anschließend auf den Eingang der bestätigenden *AckMsg*-Nachricht. Die Wartebedingung in Zeile 19 definiert dazu den *Timeout* von *1000 ms*, innerhalb dessen die Rückantwort erwartet wird, und spezifiziert die innere Wartebedingung, die auf den Eingang einer *AckMsg*-Nachricht der Gegenstelle wartet. Das Filterkriterium, das mittels eines funktionalen Lambda-Ausdrucks [71] angegeben werden kann, prüft, ob die eingehende *AckMsg* von der korrekten Gegenstelle stammt. Diese Prüfung ist besonders wichtig, wenn wie im oben angeführten Beispiel parallele Verbindungsanfragen bearbeitet werden.

Die Wartebedingung in Zeile 19 kann somit entweder durch den Eingang einer den Filter erfüllenden Nachricht oder durch den Ablauf des Zeitintervalls erfüllt werden. Die anschließende Prüfung in Zeile 21 entscheidet lediglich, welches der beiden Ereignisse eingetreten ist. Sollte der Ablauf des Zeitintervalls für die Erfüllung der Wartebedingung verantwortlich sein, so terminiert das *Handshake*-Verfahren. Andernfalls ist die Verbindung hergestellt.

Die Implementierung der *Initiator*-Rolle ist ab Zeile 27 in Auflistung 3.32 dargestellt. Die Methode *InitiateHandshake* ist ausschließlich für die Erstellung eines neuen Subautomaten zuständig. Das *Launch*-Kommando mit dem generischen Parameter *Peer* erzeugt den *InitiateHandshakeRequest*-Iterator und übergibt diesem die Adresse der zu kontaktierenden Gegenstelle.

Das folgende Verfahren innerhalb des *InitiateHandshakeRequest*-Iterator ist analog zu dem des bereits vorgestellten *HandleIncomingHandshakeRequest*-Automaten. Die *InitiateHandshakeRequest*-Methode beginnt mit dem Versand einer *SynMsg* an die Gegenstelle und wartet mit der in Zeile 35 definierten Wartebedingung entweder auf den Eingang der *SynAckMsg* der Gegenstelle oder den Ablauf des Zeitintervalls. Sollte die Gegenstelle innerhalb des definierten Intervalls geantwortet haben, vollendet der *Initiator* das Protokoll mit dem Versand der *AckMsg*-Nachricht. Die Verbindung ist im Anschluss hergestellt.

3.7.3 Existenz- und Erreichbarkeitsverfahren

Die Fragestellung der Existenz und Erreichbarkeit eines Kommunikationspartner ist für viele verteilte Anwendungen von großer Bedeutung und wird im folgenden an-

```
1   public partial class HandshakeProtocol : ProtocolBase
2   {
3       private Signal exitSignal;
4
5       public HandshakeProtocol(Scheduler scheduler)
6           : base(scheduler)
7       {
8           this.exitSignal = new Signal();
9       }
10
11      public override IEnumerator<ReceiverBase> Execute(AbstractStateMachine
            stateMachine)
12      {
13          yield return Parallel(Int32.MaxValue, (Receive<SynMsg>() +
                HandleIncomingHandshakeRequest)) | exitSignal.CreateReceiver();
14      }
15
16      private IEnumerator<ReceiverBase> HandleIncomingHandshakeRequest(
            AbstractStateMachine stateMachine, SynMsg syn)
17      {
18          SendSynAck(syn.Sender);
19          yield return Receive<AckMsg>(msg => msg.Sender == syn.Sender) |
                Timeout(1000);
20
21          if (stateMachine.CurrentMessage is TimeoutMessage)
22              yield break;
23
24          // Connection established
25      }
26
27      public void InitiateHandshake(Peer destination)
28      {
29          Launch<Peer>(InitiateHandshakeRequest, destination);
30      }
31
32      private IEnumerator<ReceiverBase> InitiateHandshakeRequest(
            AbstractStateMachine stateMachine, Peer destination)
33      {
34          SendSyn(receiver);
35          yield return Receive<SynAckMsg>(msg => msg.Sender == destination) |
                 Timeout(1000);
36
37          if (stateMachine.CurrentMessage is TimeoutMessage)
38              yield break;
39
40          SendAck(receiver);
41
42          // Connection established
43      }
44  }
```

Auflistung 3.32: Implementierung des Drei-Wege-Handshakes

hand des *Peer-To-Peer* Systems *Pastry* [104] näher erläutert. *Pastry* verfügt wie viele Systeme der dritten Generation über einen *Routing*-Mechanismus, der garantiert, dass ein *Routing*-Ziel in *O(log(n)) Overlay-Routing*-Schritten erreicht ist.

Um solch ein Verfahren zu gewährleisten muss jeder Teilnehmer über ausreichende Kenntnis des Systems verfügen und somit die Adressen von nahen sowie fernen Teilnehmern in seiner *Routing*-Tabelle speichern. Aufgrund der hohen Fluktuation der Teilnehmer muss jeder Knoten prüfen, ob die gespeicherten Kommunikationspartner noch existent und erreichbar sind. Die *Pastry*-Implementierung prüft beispielsweise in festgelegten Intervallen die Erreichbarkeit der im *Leaf Set* enthaltenen Knoten, um einen Überblick über die direkte Nachbarschaft zu erhalten. Jeder Knoten sendet dazu eine *Ping*-Nachricht an seinen Nachbarn und erwartet eine *Pong*-Nachricht als Antwort. Um die zufällige oder verspätete Zustellung einer *Pong*-Nachricht zu erkennen, wird jede *Ping*-Nachricht mit einer eindeutigen Identifizierer oder einem Zeitstempel mit begrenzter Gültigkeit versehen, der vor dem Versand der Antwort in die *Pong*-Nachricht integriert wird.

Die auf dem *Gears4Net* basierende Umsetzung dieses Verfahrens zur Existenz- und Erreichbarkeitsprüfung ist in Auflistung 3.33 dargestellt und wird im folgenden beschrieben. Die auszugsweise dargestellte partielle Klasse *PingPongProtocol*, verfügt über zwei Listen, wobei die erste die anzufragenden Knoten und die zweite die eingegangenen Antworten enthält. Darüber hinaus implementiert die Klasse neben dem initialen Iterator die Zustandsmaschinen *Processor* und *TestGenerator* und weist zudem die klassischen Merkmale eines *Gears4Net-Protocols* auf. Zu diesen zählen die Vererbungshierarchie, die initiale Zustandsmaschine *Execute* und die Übergabe eines *Schedulers* an den Basis-Konstruktor der *ProtocolBase*-Klasse.

Die Implementierung des *Ping-Pong*-Verfahrens beginnt mit der Spezifikation der *Interval*-Wartebedingung innerhalb des *Execute*-Iterators, der beim Start der *Protocol*-Instanz aufgerufen wird. Der spezifizierte und niemals terminierende *Interval-Receiver* (in Zeile 12 der Auflistung) ruft in einem Zeitintervall von *10000 ms* den *Processor*-Iterator auf. Es ist anzumerken, dass das Aufrufintervall erst dann neu gestartet wird, wenn der vorherige Aufruf des *Processor*-Iterators beendet wurde. Eine Überschneidung zweier Aufrufe ist somit ausgeschlossen.

Der *Processor* ist für die Initiierung der Erreichbarkeitsprüfung und die spätere Weiterverarbeitung der resultierenden Ergebnisse zuständig. Die Ausführung des *Launch*-Kommandos in Zeile 18 erzeugt einen *StateMachineReceiver*, der auf die Terminierung der ihm übergebenen Zustandsmaschine wartet. Im vorliegenden Fall

muss der *TestGenerator*-Automat innerhalb des Zeitintervalls von *5000 ms* terminieren, da ansonsten die Ausführung der Maschine abgebrochen wird. Unabhängig davon, welches Kriterium zur Terminierung der Wartebedingung in Zeile 18 geführt hat, erfolgt die Auswertung der Ergebnisse, die in der *responseList* abgelegt wurden. Das Terminierungskriterium gibt lediglich darüber Aufschluss, ob alle angefragten Knoten innerhalb des Zeitintervalls geantwortet haben oder nicht.

```
1   public partial class PingPongProtocol : ProtocolBase
2   {
3       private List<Peer> neighborList = new List<Peer>();
4       private List<Peer> responseList = new List<Peer>();
5
6       public PingPongAliveTestProtocol(Scheduler scheduler)
7           : base(scheduler)
8       {
9       }
10
11      public override IEnumerator<ReceiverBase> Execute(AbstractStateMachine
            stateMachine)
12      {
13          yield return Interval(Processor, 0, 10000);
14      }
15
16      public IEnumerator<ReceiverBase> Processor(AbstractStateMachine
            stateMachine)
17      {
18          yield return Launch(TestGenerator) | Timeout(5000);
19
20          foreach (Peer p in this.responseList)
21          {
22              // set alive flag
23          }
24          responseList.Clear();
25      }
26
27      public IEnumerator<ReceiverBase> TestGenerator(AbstractStateMachine
            stateMachine)
28      {
29          Guid msgID = Guid.NewGuid();
30          SendPingMsg(neighborList, msgID);
31
32          yield return Parallel(this.neighborList.Count, this.neighborList.
                Count,
33              Receive<PongMsg>(msg => msg.ID == msgID  && !this.responseList.
                    Contains(msg.Sender),
34              delegate(PongMsg msg) { this.responseList.Add(msg.Sender);}));
35      }
36  }
```

Auflistung 3.33: Implementierung des Ping-Pong Verfahrens

Die Implementierung des eigentlichen *Ping-Pong*-Verfahrens wird innerhalb des *TestGenerator*-Iterators ab Zeile 27 beschrieben. Das Verfahren beginnt mit der

Bereitstellung eines Identifizierers, der eine Testrunde eindeutig beschreibt, und dem Versand einer *Ping*-Nachricht an alle *Peers*, die in der *neighborList* verzeichnet sind. Nach dem Nachrichtenversand folgt die Spezifikation der Wartebedingung. Das ab Zeile 32 beginnende Konstrukt besteht aus einen *ParallelReceiver*, der einen *Receiver<Pong>* beinhaltet. Letzterer verfügt zudem über ein Filterkriterium und den Aufruf eines *Handlers*, der die eingegangene Nachricht weiter verarbeitet.

Eine detaillierte Betrachtung der einzelnen Elemente verdeutlicht, dass der *ParallelReceiver* in Zeile 32 maximal *neighborList.Count* viele Nachrichten verarbeitet und anschließend terminiert. Somit beendet sich der Subautomat direkt, nachdem alle angefragten Knoten geantwortet haben. Das Filterkriterium des *Receiver<Pong>* überprüft erstens, ob die eingegangene *Pong*-Nachricht über denselben Identifizierer verfügt, der initial für diese Testrunde erzeugt wurde, und zweitens, ob der Absender der *Pong*-Nachricht bereits geantwortet hat. Durch die Angabe des Filters werden falsche und doppelte Nachrichten eliminiert. Die Angabe des *Handlers*, die in Zeile 34 erfolgt, ist durch eine anonyme Methode dargestellt, der die eingehende *Pong*-Nachricht übergeben wird. Die Aufgabe des Methodenrumpfes ist lediglich das Hinzufügen der Nachrichtenabsenders zur *responseList*. Die Realisierung des *Handlers* kann ebenfalls durch die Implementierung einer Methode mit identischer Signatur erfolgen. Aufgrund der kompakten Form ermöglicht die anonyme Variante jedoch eine sehr elegante Darstellung.

3.8 Evaluation

Im Zuge der Evaluation wird das Programmiermodell *Gears4Net* sowie dessen Implementierung anhand der in Kapitel 3.2 aufgestellten Analysedimensionen für Programmiermodelle und der in Kapitel 3.3 zusätzlich definierten Anforderungen analysiert. Im folgenden Kapitel werden daher die Ressourceneffizienz des Modells sowie die Quellcodestrukturierung und die Synchronisierungseigenschaften betrachtet und mit den bestehenden Modellen verglichen. Abgeschlossen wird die Evaluation mit der Betrachtung des optimalen Verhältnisses zwischen der Anzahl der verwendeten *Protocol*-Instanzen und der Anzahl der bereitgestellten *Scheduler*-Objekte.

3.8.1 Ressourceneffizienz

Die Evaluation der Ressourceneffizienz betrachtet die Auslastung der CPU sowie die Effektivität der Nutzung des Arbeitsspeichers und leitet daraus Rückschlüsse auf

die Skalierungs- und Performanzeigenschaften des Systems ab. Die folgende Analyse betrachtet zuerst das dem *Gears4Net*-Modell zugrundeliegende Iteratorenkonzept und setzt anschließend mit der Speichereffizienzbetrachtung der *Protocol*-Instanzen fort.

Speichereffizienz des Iteratorenkonzeptes

Die Architektur des Programmiermodells *Gears4Net* sieht die Bereitstellung beliebig vieler Zustandsmaschinen innerhalb einer *Protocol*-Instanz vor, wobei jede Instanz über mindestens eine initiale Zustandsmaschine verfügt. Jede dieser Maschinen basiert auf einem Iterator-Block, der während der Kompilierung in eine Klasse umgewandelt und zur Laufzeit einer Applikation instantiiert wird. Die Umsetzung des Iteratorenkonzeptes ist somit von grundlegender Bedeutung für die Effizienz des Gesamtsystems und wird daher im Folgenden eingehend untersucht.

Die Effizienzbetrachtung des Iteratorkonzeptes kann entweder theoretisch über die Auswertung der erzeugten und innerhalb einer *Assembly* abgelegten Klassen oder praxisnah über die Messung des tatsächlichen Speicherverbrauch ausgewertet werden. Aus Gründen der Vergleichbarkeit mit den in Kapitel 3.2 durchgeführten Betrachtungen sowie der Problematik, dass bei der theoretischen Betrachtung keinerlei Aussage über das Laufzeitverhalten der Applikation sowie des *Garbage-Collectors* getroffen werden kann, erfolgt die im Folgenden beschriebene und in Abbildung 3.19 dargestellte Evaluation auf Basis einer Messung, deren Datensätze bei der Erzeugung von 45 Millionen Iteratoren gewonnen wurden.

Das Diagramm in Abbildung 3.19 wertet sowohl das *Virtual Memory (VM)* als auch das *Working Set (WS)* aus. Das *VM* beschreibt den angeforderten, das *WS* den genutzten Speicher. Neben der Betrachtung des *WS*, das ein genaues Verständnis über die Ressourcennutzung von Objekten und Algorithmen liefert, beschreibt das *VM* den vom Betriebssystem angeforderten Speicher. Die Betrachtung des *VM* ist von erheblicher Wichtigkeit, da bereits die Anforderung, nicht die Nutzung, von Speicher über die festgelegten Grenzen zu einer *OutOfMemory-Exception* führt. Bei der Darstellung der gemessenen Daten wird die Anzahl der erzeugten Iteratoren auf der horizontalen Achse des Diagramms, der Gesamtspeicherverbrauch auf der horizontalen Primärachse und der Speicherverbrauch pro Iterator auf der horizontalen Sekundärachse dargestellt.

Bei der Betrachtung der beiden Kurven, die den kumulierten Speicherverbrauch darstellen, wird deutlich, dass der Speicherverbrauch nahezu linear ansteigt und dass die Differenz zwischen angefordertem und genutztem Speicher als minimal zu

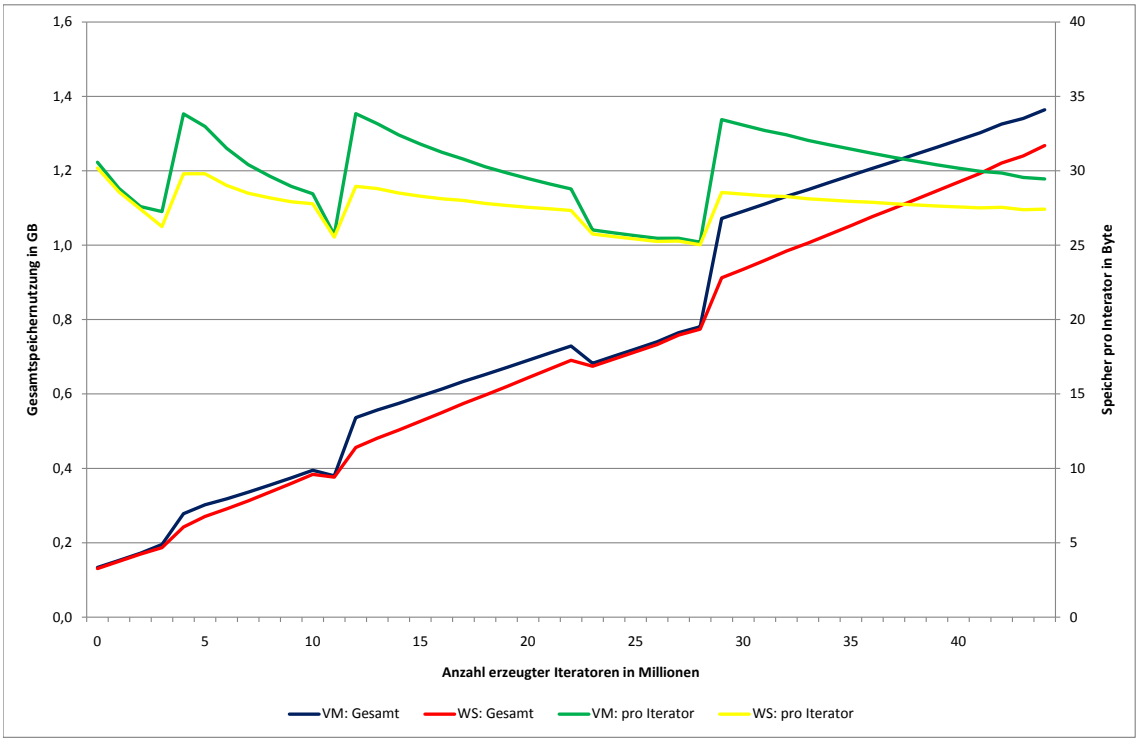

Abbildung 3.19: Speicherverbrauch bei der Erzeugung von Iteratoren (Software: Windows 7 - 32 Bit, Hardware: Lenovo Thinkpad T60p 2007-8HG, 2 GB RAM)

betrachten ist. Die in der Kurve abzulesenden, wiederkehrenden Unregelmäßigkeiten sind auf das Einwirken des automatischen Speichermanagements sowie den Einsatz des *Garbage-Collectors* zurückzuführen. Die Verlauf der beiden Kurven, die den Speicherverbrauch pro Iterator-Instanz angeben, ist aufgrund des nahezu linearen Verhaltens des kumulierten Speichers als konstant zu werten. Eine Iterator-Instanz verbraucht im Durchschnitt *25* bis *30 Byte* Arbeitsspeicher und ist somit extrem kosteneffizient.

Zusammenfassend lässt sich anmerken, dass die Verwendung des Iteratorenkonzeptes einen Aufwand erzeugt, der aufgrund der effizienten Implementierung der Iteratoren und der minimalen Speicheranforderungen zu vernachlässigen ist.

Speichereffizienz der Protocol-Instanzen

Das Speichernutzungsverhalten einer *Gears4Net*-Applikation ist abhängig von der Anzahl der erzeugten *Protocol*-Instanzen, sowie der Anzahl der Zustandsmaschinen pro Instanz und der Komplexität der definierten Wartebedingungen. Neben dem strukturellen Aufbau der *Protocol*-Instanzen spielt die Menge der verwendeten *Sche-*

duler eine entscheidende Rolle, da jeder *Scheduler* über einen eigenen *Thread* mit einem *Stack* von jeweils *512 KB* Größe verfügt.

Die Evaluation des Speichernutzungsverhaltens des *Gears4Net*-Implementierung erfolgt analog zum Iteratorenkonzept anhand einer Messung, bei der jeweils eine Million *Protocol*-Instanzen in verschiedenen Konfigurationen erzeugt wurden. Die verschiedenen Konfigurationen unterscheiden sich in der Anzahl der verwendeten Zustandsmaschinen pro *Protocol*-Instanz. Hierbei wird zwischen einer und drei Zustandsmaschinen variiert, die jeweils über identische Wartebedingungen verfügen. Die zweite Varianz erfolgt in der Anzahl der verwendeten *Scheduler*. Diese beläuft sich je nach Messung zwischen 1, 10 und 100 *Schedulern*, auf die die *Protocol*-Instanzen gleichmäßig verteilt wurden.

Das Resultat der Messungen ist in den Diagrammen 3.20 und 3.21 dargestellt, wobei das erste Diagramm die Nutzung des virtuellen Speichers darstellt und in der zweiten Abbildung die Messwerte für den belegten Speicher verzeichnet sind. Die jeweils sechs Kurven der Diagramme sind mit der Anzahl der verwendeten *Scheduler* und sowie der Anzahl der Zustandsmaschinen pro *Protocol*-Instanz bezeichnet. Die Ergebnisse des ersten Diagramms verdeutlichen, dass der virtuelle Speicher für alle Messungen mit zunehmender Anzahl an *Protocol*-Instanzen linear steigt und keine exponentiellen Tendenzen aufzeigt. Zudem ist erkennbar, dass die Kurven mit gleicher Anzahl an Zustandsmaschinen, aber unterschiedlicher Anzahl *Scheduler*-Objekten lediglich um einen konstanten Faktor verschoben sind. Die Größe des konstanten Faktors ist abhängig von der Anzahl der *Scheduler* und der damit verbundenen Anzahl an verwendeten *Threads*. Verwendet eine Applikation beispielsweise 100 *Scheduler*-Objekte mehr als eine zweite, verschiebt sich die resultierende Diagrammkurve um den Faktor *100 * (512 KB (Thread-Stack-Size) + (Scheduler Overhead) + (Thread Verwaltung)*. Das Ergebnis einer solchen Verschiebung wird beim Vergleich der Kurven *(S = 1, SM = 3)* und *(S = 100, SM = 3)* deutlich. Die zweite Kurve liegt dabei circa *100 MB* über der Kurve mit nur einem *Scheduler*.

Neben der Allokation von virtuellem Speicher ist die reale Speichernutzung von entscheidender Bedeutung. Die Diskrepanz dieser beiden Eigenschaften entsteht dadurch, dass beispielsweise für einen *Thread* ein *Stack* mit der Größe von *512 KB* virtuellem Speicher allokiert wird und der *Thread* diesen allokierten Speicher nicht vollständig ausnutzt und somit der reale Speicherverbrauch unter dem angeforderten Limit liegt.

Betrachtet man mit diesem Hintergrund das in Abbildung 3.21 dargestellte Diagramm, so wird deutlich, dass die Kurven mit einer identischen Anzahl an Zustands-

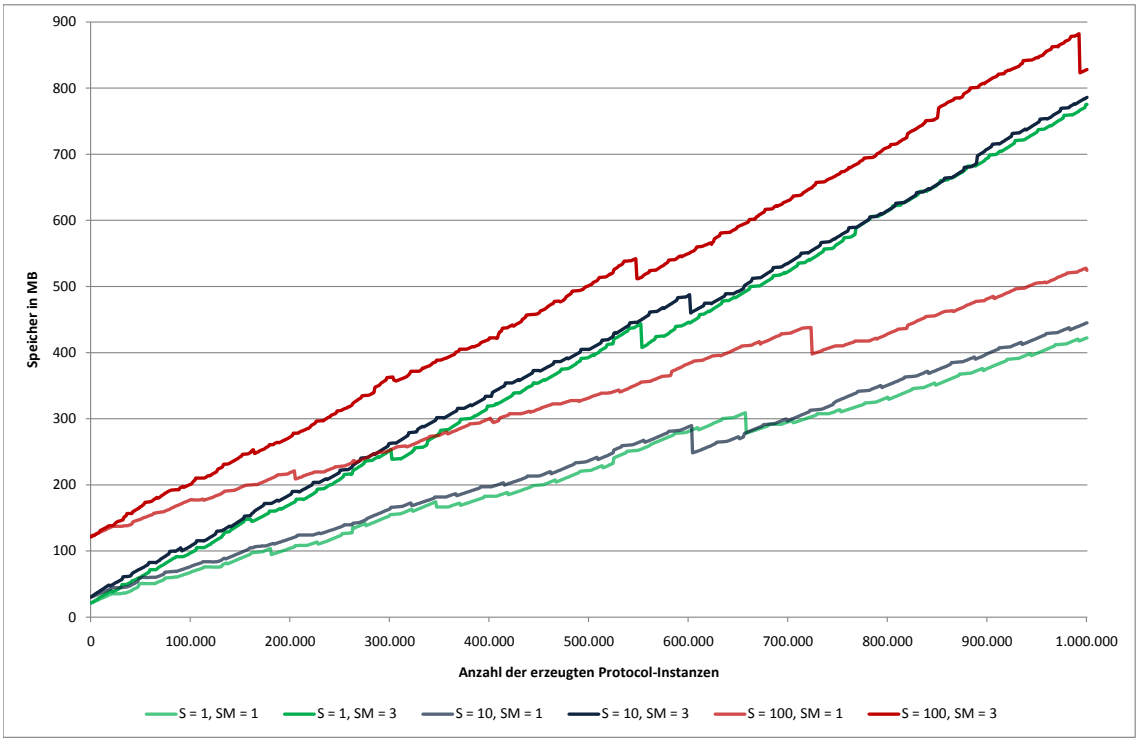

Abbildung 3.20: Virtueller Speicherverbrauch (VM) bei der Erstellung von Protocol-Instanzen mit 1 bzw. 3 State-Machines (SM), sowie 1, 10 oder 100 Schedulern (S) (Software: Windows 7 - 32 Bit, Hardware: Lenovo Thinkpad T60p 2007-8HG, 2 GB RAM)

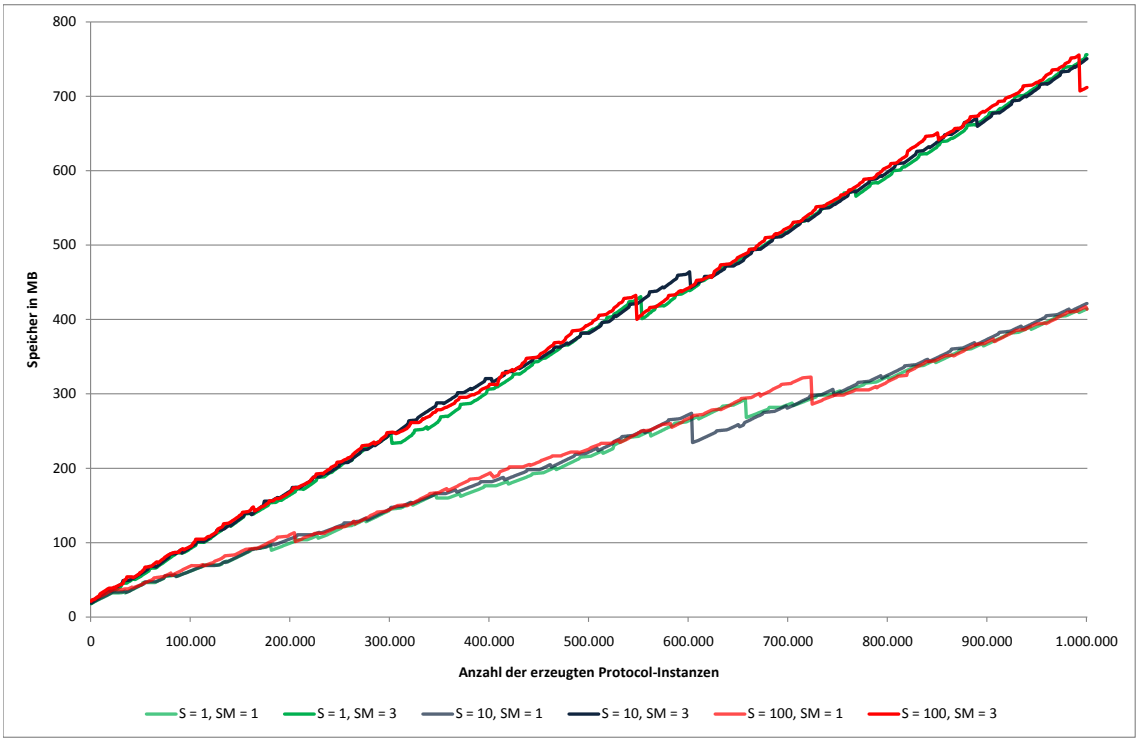

Abbildung 3.21: Speicherverbrauch (WS) bei der Erstellung von Protocol-Instanzen mit 1 bzw. 3 State-Machines (SM), sowie 1, 10 oder 100 Schedulern (S) (Software: Windows 7 - 32 Bit, Hardware: Lenovo Thinkpad T60p 2007-8HG, 2 GB RAM)

maschinen und unterschiedlicher *Scheduler*-Anzahl nahezu deckungsgleich sind. Die Anzahl der verwendeten *Scheduler*-Objekte wirkt sich somit nicht signifikant auf den verwendeten Speicher aus. Ausgehend von den linearen Kurvencharakteristiken der Diagramme 3.20 und 3.21 kann von einem konstanten Speicherverbrauch pro *Protocol*-Instanz ausgegangen werden, was durch die Auswertung in Abbildung 3.22 unterstrichen wird.

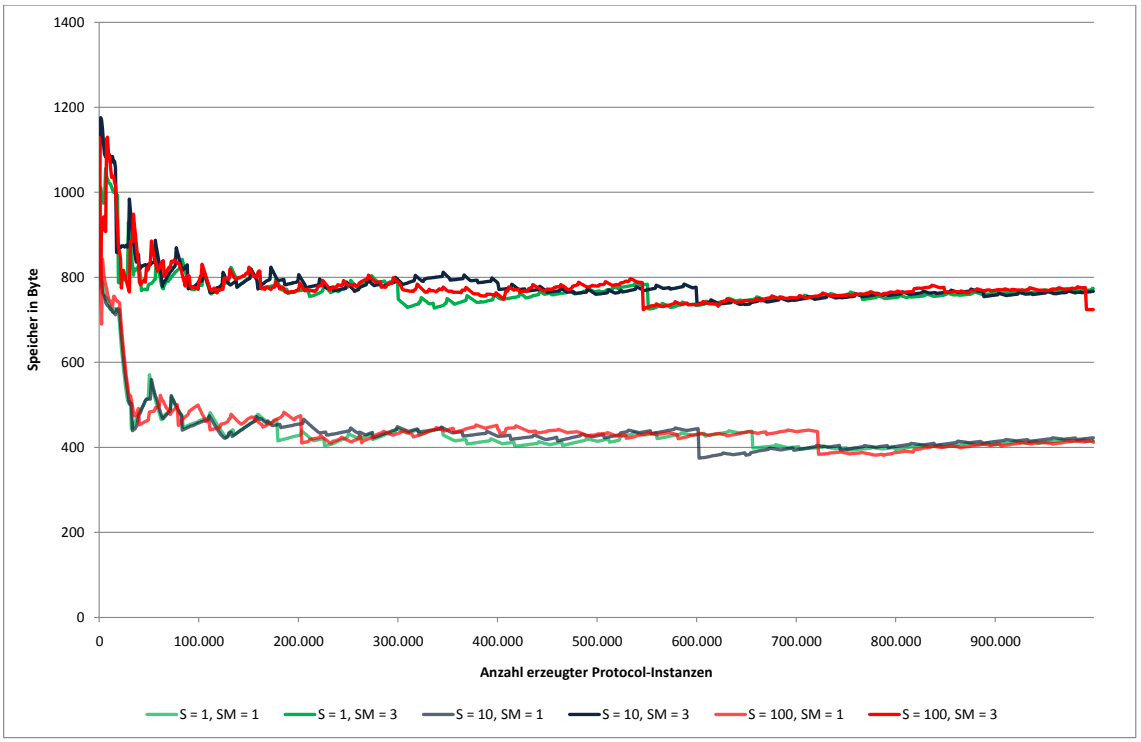

Abbildung 3.22: Speicherverbrauch pro Protocol-Instanz mit 1 bzw. 3 State-Machines (SM), sowie 1, 10 oder 100 Schedulern (S) (Software: Windows 7 - 32 Bit, Hardware: Lenovo Thinkpad T60p 2007-8HG, 2 GB RAM)

Zusammenfassend betrachtet folgt das beschriebene Messergebnis den Tendenzen des Iteratorenkonzeptes. Die *Gears4Net*-Implementierung stellt mit den *Protocol*-Instanzen ein sehr skalierbares Konzept bereit, bei dem problemlos mehrere Millionen parallel ausführbarer Instanzen erzeugt werden können, ohne die Speichereffizienz des Systems zu beeinflussen. Zudem zeigt sich keine signifikante Einschränkung bezüglich der Speichereffizienz bei der Verwendung einer größeren Menge an *Scheduler*-Instanzen.

3.8.2 Quellcodestrukturierung

Der Aspekt der Strukturierung des Quelltextes umfasst die Zustandsmaschinen-ähnliche Programmierung sowie den Verzicht auf klassische *Callback-Handler* und *Polling*-Schleifen. Der zweite Aspekt, die Erkennung komplexer Ereignisse, betrachtet, inwieweit sich die Erkennung in das bestehende Programmiermodell integriert oder Modellabweichungen erfordert.

Die Evaluation beider Aspekte erfolgt anhand des in Auflistung 3.34 dargestellten Quellcodebeispiels. Der Aufbau des Quelltextes folgt vollständig dem Konzept der Zustandsmaschinen-ähnlichen Programmierung und präsentiert sich nahezu äquivalent zum Modell der blockierenden Programmierung. Unterschieden werden die Modelle lediglich durch die Eigenschaft, dass *Gears4net*-basierende Programme nicht blockieren, sondern in ihrer Ausführung vom *Scheduler* unterbrochen werden. Die Punkte der möglichen Unterbrechungen werden durch die Angabe der Schlüsselwörter *yield-return* beziehungsweise *yield-break* gekennzeichnet, wobei letzteres auf das Verlassen der Methode hinweist.

```
1       public override IEnumerator<ReceiverBase> Execute(AbstractStateMachine sM)
2       {
3           yield return Receive<A>(msg => msg.ID == 42) + HandleMessageA;
4
5           yield return Receive<B>() | Receive<C>() | Timeout (1000);
6       }
7
8       private IEnumerator<ReceiverBase> HandleMessageA(AbstractStateMachine sM, A
            msg)
9       {
10          // Implement sub state-machine
11      }
```

Auflistung 3.34: Strukturierung des Quelltextes im Gears4Net-Programmiermodell

Das Modell verzichtet auf die Nutzung der aus dem asynchronen Programmiermodell bekannten Rückrufmethoden, erlaubt jedoch die Strukturierung des Quellcodes in Subroutinen beziehungsweise in Zustandsmaschinen durch die Anwendung des *PLUS*-Operators. Das dritte Kriterium des ersten Analyseaspektes bildet der Verzicht auf jegliche *Polling*-Mechanismen. Diesem Verzicht kommt das *Gears4Net*-Programmiermodell vollständig nach, da das Modell ausschließlich auf dem Modell der asynchronen Programmierung aufsetzt und somit keinerlei Berührungspunkte mit *Polling*-Konzepten bestehen.

Der Erkennung und Darstellung von komplexen Ereignissen ist das *Gears4Net*-Programmiermodell durch die Einführung der *Receiver*-Wartebedingungen nachgekommen. Die Wartebedingungen, die sich vollständig in das Programmiermodell integrieren, bieten dem Entwickler die Möglichkeit, verschiedenste, unabhängige Wartebedingungen miteinander zu verknüpfen. Das Beispiel in Auflistung 3.34 verdeutlicht dies anhand der in den Zeilen 3 und 5 aufgezeigten Anweisungen. Die Spezifikation der ersten Wartebedingung setzt sich aus zwei Komponenten zusammen. Die erste Komponente besteht aus einer *Receiver*-Wartebedingung, die auf den Eingang einer Nachricht des Typs *A* wartet, die dem angegebenen Filterkriterium *(ID = 42)* entspricht. Die zweite Komponente des Gesamtausdrucks wird durch die Konkatenation mit dem Funktionszeiger *HandleMessageA* auf den in Zeile 8 dargestellten Iterator vollendet.

Ein weiteres, in der Praxis sehr geläufiges Anwendungsbeispiel der komplexen Wartebedingung ist in Zeile 5 aufgezeigt. Der Gesamtausdruck evaluiert *wahr*, sobald eines der drei disjunkten Ereignisse eintritt. Dies ist also genau dann der Fall, wenn Nachrichten der Typen *B* beziehungsweise *C* eingehen oder das Zeitintervall von *1000 ms* abgelaufen ist.

Zusammenfassend betrachtet genügt das *Gears4Net*-Programmiermodell bezüglich der Quellcodestrukturierung allen im Kapitel Anforderungsanalyse gestellten Aspekten und Kriterien an ein Programmiermodell für massiv parallele und verteilte System, indem es ein intuitives und Zustandsmaschine-ähnliches Modell mit der Erkennung von komplexen Ereignissen verknüpft.

3.8.3 Synchronisierung

Die Architektur des *Gears4Net*-Programmiermodells setzt auf in Kapitel 3.2.4 vorgestellten Aktorenmodell auf, wobei jeder Aktor von einer eigenen Recheneinheit ausgeführt wird. Die Umsetzung dieses Konzeptes erlaubt somit die synchronisierungsfreie Ausführung der *Gears4Net Protocol*-Instanzen durch einen *Scheduler*.

3.8.4 Ausführungsumgebung der Protocol-Instanzen

Das Verhältnis zwischen *Protocol*-Instanzen und *Schedulern* betrachtet die Auswirkungen der Ausführungsgeschwindigkeit bei unterschiedlicher Anzahl *Protocol*-Instanzen, die auf einem *Scheduler*-Objekt ausgeführt werden. Die zweite Relation zwischen der Anzahl der *Scheduler* und der bereitgestellten Hardwareinfrastruktur

beschäftigt sich mit der Frage, ob es eine optimale Anzahl *Scheduler* für eine vorgegebene Hardwaretopologie gibt.

Die Evaluation dieser beiden Fragestellungen erfolgt mittels einer Messung unter absoluter Volllast. Das Messverfahren basiert auf der Idee, die Ausführungsgeschwindigkeit in Form von verarbeiteten Nachrichten auf unterschiedlichen *Protocol*- und *Scheduler*-Konfigurationen in einem definierten Zeitintervall zu betrachten. Das für die Messung verwendete *Protocol* ist in Auflistung 3.35 aufgezeigt. Es entspricht einer großen und immer wiederkehrenden Schleife, bei der das *Protocol* jeweils eine Nachricht in seine eigene Warteschleife hinzufügt und im Anschluss auf den Eingang dieser Nachricht wartet. Das *Parallel*-Kommando in Zeile 14 wartet dazu auf eine unbegrenzte Anzahl an *Msg*-Nachrichtenobjekten und führt im Anschluss den *Send*-Iterator in Zeile 17 aus. Dieser speichert den Zeitpunkt des Nachrichteneingangs, fügt die verarbeitete Nachricht wieder in die Warteschleife ein und startet somit den wiederkehrenden Zustellungsprozess.

```
1   public class ThroughputTest : ProtocolBase
2   {
3       public List<long> ReceivedMessages { get; }
4
5       public ThroughputTest(Scheduler scheduler)
6           : base(scheduler)
7       {
8           this.ReceivedMessages = new List<long>();
9       }
10
11      public override IEnumerator<ReceiverBase> Execute(AbstractStateMachine
            stateMachine)
12      {
13          BroadcastMessage(new Msg());
14          yield return Parallel(Int32.MaxValue, Receive<Msg>(null, Send)) |
                Termination();
15      }
16
17      private void Send(Msg msg)
18      {
19          this.ReceivedMessages.Add(Environment.TickCount);
20          BroadcastMessage(msg);
21      }
22  }
```

Auflistung 3.35: Protocol zum Test des Nachrichtendurchsatzes

Die Messung besteht aus allen Permutationen zwischen *Protocol*-Instanz und *Scheduler*-Konfiguration und wurde auf einer Maschine mit zwei beziehungsweise vier Rechenkernen durchgeführt. Der Wertebereich der *Protocol*-Instanzkonfiguration

lässt alle Zweierpotenzen zwischen *1* bis *32768*, der Wertebereich der *Scheduler*-Konfiguration alle Zweierpotenzen von *1* bis *64* *Scheduler*-Instanzen zu. Die Ergebnisse der insgesamt *77* Messungen pro *Hardwarearchitektur* sind in den Diagrammen 3.23 und 3.24 dargestellt. Für die Evaluation der Resultate wurden die *ReceivedMessages*-Listen der ausgeführten *Protocol*-Instanzen im Anschluss an die Messungen konkateniert und dahingehend gruppiert, dass der Nachrichtendurchsatz des Gesamtsystems pro Sekunde abgelesen werden kann.

Das vorgestellte Messverfahren betrachtet die maximalen Einflüsse zweier *Scheduling*-Verfahren auf eine *Gears4Net*-basierende Applikation. Den ersten Einflussfaktor bildet die Variation der Anzahl der *Protocol*-Instanzen, den zweiten die Anzahl der *Scheduler*-Objekte. Die Variation der Anzahl der *Protocol*-Instanzen führt zur Änderung des *Scheduling*-Aufwandes des *Gears4Net-Schedulers*, da die Steigerung der *Protocol*-Anzahl pro *Scheduler* einen häufigeren Wechsel zwischen den Instanzen erfordert. Der zweite Einflussfaktor, die Anzahl der verwendeten *Gears4Net-Scheduler*-Instanzen, verlagert die Fragestellung des optimalen *Schedulings* auf das Betriebssystem, da jeder *Gears4Net-Scheduler* über einen eigens vom System bereitgestellten *Thread* verfügt, der durch die *Scheduling*-Algorithmen bedient werden muss.

Die im Folgenden dargestellten und beschriebenen Ergebnisse resultieren aus der Messung auf einer Maschine mit zwei Rechenkernen. Auf die Darstellung der Messdaten der Vier-Kern-Maschine wurde aufgrund der identischen Aussagekraft verzichtet. Abbildung 3.23 trägt die Anzahl der *Protocol*-Instanzen pro *Scheduler* auf der horizontalen und die Anzahl der verarbeiteten Nachrichten auf der vertikalen Achse auf. Das resultierende Diagramm zeigt einen optimalen Nachrichtendurchsatz bei 128 *Protocol*-Instanzen, die zu jeweils 4 Einheiten auf 32 *Schedulern* ausgeführt werden. Das hauptsächliche Interesse der Messung liegt jedoch weniger auf dem vorgestellten Optimum sondern vielmehr auf der Fragestellung, inwieweit die Performanz des Systems einbricht, falls eine *Gears4Net*-Applikation nicht in der optimalen Konfiguration ausgeführt wird. Unter der Prämisse einer Durchsatzminderung von *10%* beziehungsweise *20%* erlaubt das System einen Toleranzbereich bezüglich der Anzahl der *Protocol*-Instanzen pro *Scheduler* zwischen *1* und *32* für den ersten, beziehungsweise zwischen *1* und *64* für den zweiten Fall. Der Toleranzbereich der Anzahl der verwendeten *Scheduler*-Instanzen ist für eine Durchsatzminderung von *10%* für *2* bis *62*, für eine Durchsatzminderung von maximal *20%* für *2* bis *64* *Scheduler*-Instanzen festzulegen.

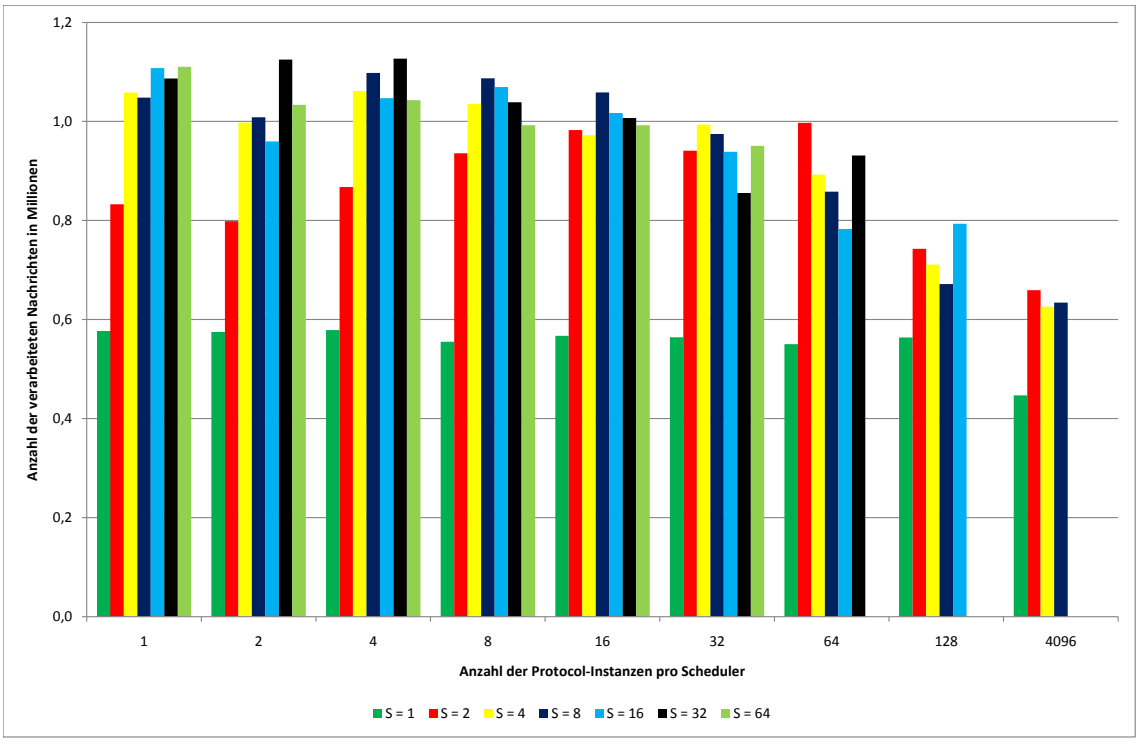

Abbildung 3.23: Nachrichtenverarbeitung pro Sekunde in Abhängigkeit von der Anzahl der Protocol-Instanzen (PI) pro Scheduler (S)

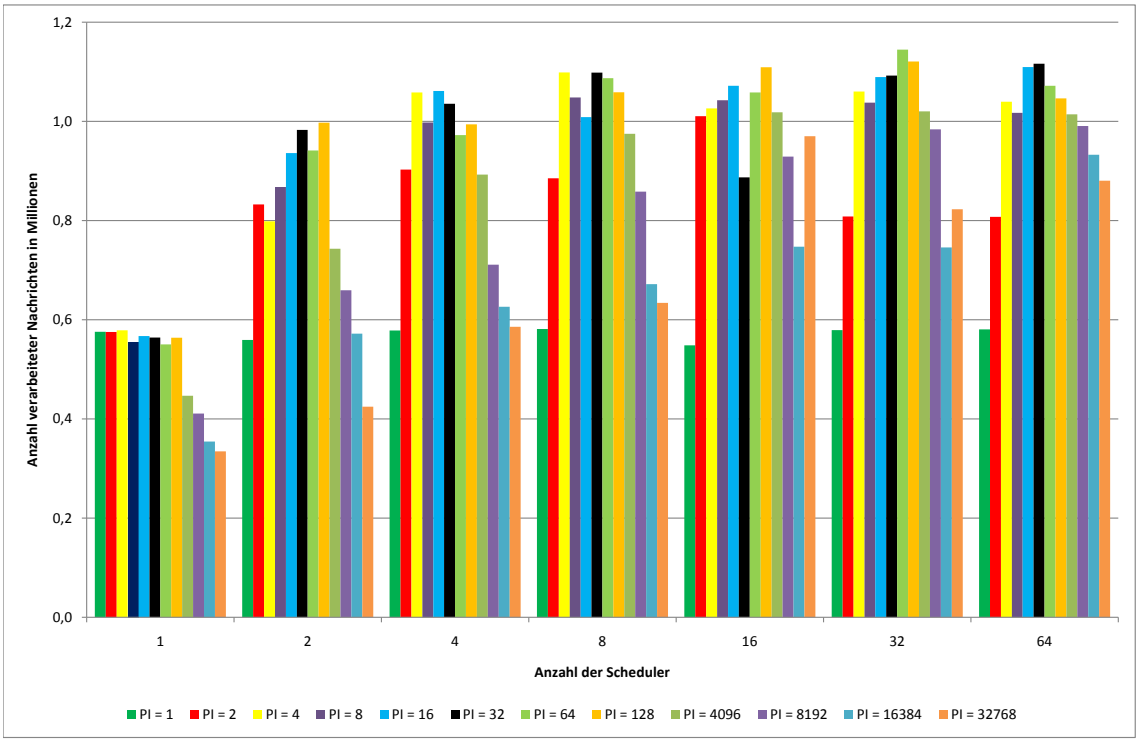

Abbildung 3.24: Nachrichtenverarbeitung pro Sekunde in Abhängigkeit von der Anzahl der Scheduler sowie der Gesamtanzahl der Protocol-Instanzen (PI)

Die in Abbildung 3.24 dargestellten Ergebnisse verdeutlichen die Aussage der in Diagramm 3.23 aufgezeigten Messergebnisse. Abbildung 3.24 trägt die Anzahl der *Scheduler*-Instanzen auf der horizontalen Achse auf und betrachtet den Nachrichtendurchsatz für die Anzahl der *Protocol*-Instanzen des Gesamtsystems. Die Abbildung verdeutlicht, dass die Anwendung ab einer Anzahl von vier *Schedulern* in den Bereich der optimalen Ausführungsgeschwindigkeit eintritt.

Die vorgestellten Messergebnisse wurden, wie bereits ausgeführt, durch eine Messung auf einer Maschine mit zwei Rechenkernen gewonnen. Die Auswirkungen dieser Konfiguration sind sowohl in Abbildung 3.23 als auch in Abbildung 3.24 ersichtlich, da der Nachrichtendurchsatz der Messung mit genau einer *Scheduler*-Instanz deutlich hinter den weiteren Messungen zurückbleibt. Durch den Einsatz einer *Scheduler*-Instanz steht lediglich ein *Thread* für die Ausführung der *Protocol*-Instanzen bereit. Dieser kann ausschließlich auf einem Kern der Maschine ausgeführt werden. Der zweite Kern trägt somit nicht zum Durchsatz des Systems bei. Die durchgeführten Messungen offenbaren daher, dass für ein System mit optimalem Durchsatz mindestens eine *Scheduler*-Instanz pro CPU-Rechenkern bereitgestellt werden sollen.

3.9 Weiterentwicklung des Programmiermodells Gears4Net

Das *Gears4Net*-Programmiermodell bildet die Grundlage für die synchronisierungsfreie Entwicklung von massiv parallelen und verteilten Anwendungen. Das Modell fordert jedoch die Einhaltung der zentralen Regel, die besagt, dass zwei *Protocol*-Instanzen ausschließlich über unveränderbare Nachrichten kommunizieren und dass keinerlei blockierende Funktionsaufrufe durchgeführt werden dürfen.

Da das *Gears4Net*-Modell ausschließlich eine *API* bereitstellt, die auf der Programmiersprache *C#* sowie dem Microsoft .NET Framework 3.5 basiert und keine *Compiler*-Erweiterung implementiert, kann die Einhaltung dieser Regel nicht automatisiert überprüft werden. Die Modifikation des *C#*-Compilers hat direkte Auswirkungen auf die zur Verfügung stehende Entwicklungsumgebung sowie auf die auf den aktuellen *C#*-Compiler zugeschnittenen *Debug*-Mechanismen. Das folgende Konzept sieht daher die Entwicklung einer *C#*-Sprach- und Compilererweiterung vor, die als *Pre*-Compiler dem eigentlichen *C#*-Compiler vorgeschaltet werden. Dieses Verfahren ermöglicht gleichzeitig die Erkennung von Programmierfehlern und zeigt zudem keinerlei Auswirkungen auf die Entwicklungsumgebung und *Debug*-Mechanismen.

Der Ansatz zerlegt die vorgestellte Regel in seine drei elementaren Bestandteile. Er betrachtet die Unveränderbarkeit der Nachrichten unabhängig von der Fragestellung, ob zwei *Protocol*-Instanzen ausschließlich über Nachrichtenaustausch und nicht über den direkten Aufruf von Funktionen miteinander kommunizieren, und stellt sicher, dass keinerlei blockierende Systemaufrufe durchgeführt werden können.

3.9.1 Unveränderbare Nachrichten

Die im Kapitel 3.6.1 vorgestellten Nachrichten des *Gears4Net*-Programmiermodells implementieren die Schnittstellen *IMessage*. Dies ermöglicht die Verbreitung der Nachrichtenobjekte über die ebenfalls in Kapitel 3.6.1 beschriebenen Warteschlangen. Insofern die verbreiteten Nachrichten die Grenzen eines Prozesses überschreiten und somit unweigerlich serialisiert werden, sind mögliche *Race Conditions* ausgeschlossen. Das Phänomen der *Race Conditions* kann jedoch dann auftreten, wenn eine Nachricht in die Warteschlange eingestellt wird und entweder der Sender oder der Empfänger schreibend auf die Nachrichteninstanz zugreift, während der andere Kommunikationspartner einen lesenden Zugriff auf der identischen Instanz vornimmt.

Betrachtet man die in Auflistung 3.36 dargestellt Standardnachricht, so wird deutlich, dass eine Änderung des Eigenschaftsfeldes *MyProperty* direkten Einfluss auf den Sender sowie den Empfänger der Nachricht hätte, da sowohl die Referenz des Senders als auch die des Empfängers auf ein und dieselbe Instanz zeigen.

```
1   public class MyMessage : IMessage
2   {
3       public int MyProperty { get; set;}
4
5       public MyMessage(int myProperty)
6       {
7           this.MyProperty = myProperty;
8       }
9   }
```

Auflistung 3.36: Gears4Net Standardnachricht

Die Bewältigung dieser Problemstellung bedingt die Effizienz des späteren Laufzeitverhaltens. Eine erste, relativ intuitive, dafür ineffiziente Lösung würde die Kopie aller Nachrichten, inklusive der enthaltenen Datenstrukturen, bieten. Die Vorteile dieses sehr einfachen Verfahren würden durch die große Nachteile überschattet, die

sich durch den immensen Speicherverbrauch sowie die hohe Rechenleistung auszeichnen.

Der in diesem Abschnitt vorgeschlagene Ansatz nutzt hingegen die als *Copy-On-Write* [5, 44] beziehungsweise *Implicit Sharing* [33] bezeichneten Konzepte, die ein Nachrichtenobjekt nur dann duplizieren, wenn ein schreibender Zugriff erfolgt. Die Implementierung von Nachrichtenklassen, die das *Copy-On-Write*-Konzept implementieren ist jedoch im Vergleich zur Implementierung einer Standardnachricht relativ aufwändig und sollte daher keinem Entwickler zugemutet werden.

Eine Spracherweiterung könnte dieser Problemstellung entgegentreten und analog zum in Kapitel 2.6 vorgestellten Iterator-Verfahren automatisiert Quellcode erstellen, sobald ein Schlüsselwort erkannt wurde. Die Einführung des Schlüsselwortes *message* in die Sprache *C#* ermöglicht somit die sehr kompakte Darstellung von Nachrichten, die auf dem *Copy-On-Write*-Konzept basieren. Die Implementierung einer solchen Nachricht, die mit Ausnahme des Schlüsselwortes *message* der Syntax der Programmiersprache *C#* entspricht, ist in Auflistung 3.37 verzeichnet.

```
1       public message MyMessage
2       {
3           public int MyProperty { get; set;}
4       }
```

Auflistung 3.37: Gears4Net Nachricht mit dem Schlüsselwort message

Durch die Spracherweiterung erlaubt der modifizierte *Compiler*, die *message*-Klasse in eine standardkonforme und das *Copy-On-Write*-Konzept implementierende *C#*-Klasse zu überführen und diese innerhalb einer Microsoft .NET *Assembly* abzulegen. Die Auflistung 3.38 zeigt die Implementierung des *Copy-On-Write*-Konzepts. Dabei werden zwei ineinander verschachtelte Klassen erzeugt, wobei die äußere den Namen der *message*-Klasse, die innere Klasse den Namen mit vorgesetztem Präfix *Inner* trägt. Die hierarchische Klassenstruktur folgt der Idee, dass alle Entitäten, die eine Referenz auf die Nachricht halten möchten, auf unterschiedliche *MyMessage*-Objekte referenzieren, die wiederum alle auf die identischen *InnerMyMessage*-Instanz referenzieren, die die eigentlichen Nutzdaten der Nachricht enthält. Jeder der *MyMessage*-Objektinstanzen hält solange an der Referenz zur gemeinsamen *InnerMyMessage*-Instanz fest, bis der erste schreibende Zugriff erfolgt. Bevor die zu schreibenden Daten in der *InnerMyMessage*-Instanz abgelegt werden, wird diese kopiert, und die anstehenden Modifikationen werden an der lokalen Kopie der *InnerMyMessage*-Instanz vorgenommen.

Die Auswirkungen eines schreibenden Zugriffs sind daher lokal beschränkt, wodurch die Entstehung von Nebenläufigkeitseffekten verhindert wird. Zudem arbeitet das Verfahren sehr effizient, da das kostenintensive Kopieren der Nutzdaten nur dann durchgeführt wird, wenn Daten modifiziert wurden. Das Duplizieren der *MyMessage*-Objekthüllen ist nahezu zu vernachlässigen, da die Instanzen ausschließlich eine Referenz auf eine Instanz der *InnerMyMessage*-Klasse verfügen.

Die Implementierung des *Copy-On-Write* Verfahrens wird am Beispiel der *MyMessage*-Nachrichtenklassen in Auflistung 3.38 dargestellt. Die *InnerMyMessage*-Klasse, deren Quelltext ab Zeile 3 aufgezeigt wird, verfügt über zwei Konstruktoren, eine Methode sowie das Datenfeld, das in Auflistung 3.37 definiert wurde. Der öffentliche Standardkonstruktor wird zur Erzeugung einer neuen Instanz, der private Konstruktor zum Erstellen einer Kopie verwendet und ausschließlich aus der Methode *CloneInnerMessage* aufgerufen. Die *MyMessage*-Klasse verfügt neben der inneren Klasse über zwei private Felder, ein durchgreifendes *Property*, sowie über zwei Konstruktoren und die Methode *GetSharedCopy*. Das erste der beiden Felder speichert die Referenz auf die Instanz der *InnerMyMessage*-Klasse, das Feld *isSent* stellt sicher, ob eine bereits Nachricht versandt wurde. Die Semantik der beiden Konstruktoren folgt analog denen der *InnerMyMessage*-Klasse.

Sobald eine Nachricht in die *BroadcastQueue* einer *Protocol*-Instanz eingestellt wird, gilt diese als versandt. Die *Protocol*-Instanz fügt jedoch nicht das übergebene *MyMessage*-Objekt sondern eine durch den Aufruf der Methode *GetSharedCopy* erzeugte Kopie der Nachrichtenhülle zur *BroadcastQueue* hinzu. Die Kopie der Nachrichtenhülle verweist analog zum Originalobjekt auf die identische Instanz der *InnerMyMessage* und kennzeichnet die Nachricht durch das Setzen des *isSent*-Feld in Zeile 51 als versandt.

Der Zugriff auf die Datenfelder der Nachrichtenklasse erfolgt über das in den Zeilen 25 bis 37 dargestellte *Property*. Der lesende Zugriff erfordert ein direktes Durchgreifen auf das gleichnamige *Property* der *InnerMyMessage*-Instanz. Erfolgt ein schreibender Zugriff, muss geprüft werden, ob vor Ablage des zu schreibenden Datums eine Kopie der *InnerMyMessage*-Instanz erzeugt werden muss. Das Kopierverfahren wird genau dann eingeleitet, wenn das Feld *isSent* dem Wert *wahr* entspricht. Innerhalb des *Setter*-Blocks des *Properties* wird dazu in Zeile 32 die Methode *CloneInnerMessage* aufgerufen, die der Klassenvariable *innerMsg* die Referenz auf die erstellte Kopie zuweist. Ab diesem Zeitpunkt verfügt die Instanz über ein eigenes *InnerMyMessage*-Objekt, dessen Änderungen ausschließlich lokale Relevanz haben.

```
1   public class MyMessage
2   {
3       private class InnerMyMessage
4       {
5           public int MyProperty { get; set; }
6
7           public InnerMyMessage()
8           {
9           }
10
11          private InnerMyMessage(InnerMyMessage clone)
12          {
13              this.MyProperty = clone.MyProperty;
14          }
15
16          public MyMessageIntern CloneInnerMessage()
17          {
18              return new InnerMyMessage(this);
19          }
20      }
21
22      private MyMessageIntern innerMsg;
23      private bool isSent = false;
24
25      public int MyProperty
26      {
27          get { return this.innerMsg.MyProperty; }
28          set
29          {
30              if (this.isSent)
31              {
32                  this.innerMsg = innerMsg.CloneInnerMessage();
33                  this.isSent = false;
34              }
35              this.innerMsg.MyProperty = value;
36          }
37      }
38
39      public MyMessage(int myProperty)
40      {
41          this.innerMsg = new InnerMyMessage() { MyProperty = myProperty };
42      }
43
44      private MyMessage(InnerMyMessage innerMyMessage)
45      {
46          this.innerMsg = innerMyMessage;
47      }
48
49      public MyMessage GetSharedCopy()
50      {
51          this.isSent = true;
52          return new MyMessage(this.innerMsg);
53      }
54  }
```

Auflistung 3.38: Implementierung des Copy-On-Write Konzepts

Innerhalb des *Setter*-Blockes wird zudem das *isSent*-Feld zurück auf *falsch* gesetzt und im Anschluss mit dem Zuweisen des zu speichernden Datums fortgefahren.

Die Implementierung des *Copy-On-Write*-Konzepts erlaubt in Verbindung mit einer Erweiterung der Sprache *C#* eine sehr speicher- und CPU-effiziente Implementierung, die jegliche *Race Conditions* ausschließt.

3.9.2 Restriktion auf Nachrichten-basierte Kommunikation

Im vorangegangenen Abschnitt wurde die Unveränderbarkeit der Nachrichtenklassen durch die Implementierung des *Copy-On-Write*-Konzepts sichergestellt. Die Umsetzung dieses Verfahrens garantiert, dass zwei *Protocol*-Instanzen niemals beschreibbaren Speicher teilen. Das gemeinsame Lesen von unveränderbaren Nachrichtenobjekten über *Protocol*- beziehungsweise *Thread*-Grenzen ist hingegen sicher und schließt mögliche *Race Conditions* aus.

Nachdem Ausschluss von *Race Conditions* durch die Verwendung von unveränderbaren Nachrichtenobjekten muss sichergestellt werden, dass die Kommunikation mit einer *Protocol*-Instanz beziehungsweise zwischen zwei Instanzen ausschließlich Nachrichten-basiert vollzogen wird. Dieses stellt aus zwei Gründen eine Herausforderung an die Spracherweiterung dar. Erstens kann ein beliebiges Objekt beziehungsweise eine *Protocol*-Instanz eine Referenz auf ein weiteres *Protocol* halten und Methoden oder *Properties* über die öffentliche Schnittstelle der *Instanz* aufrufen. Zweitens kann eine *Protocol*-Klasse über statische Klassenfelder verfügen, die per Definition von jeder Instanz einer *Protocol*-Klasse innerhalb eines Prozessraums geteilt werden.

Der Zugriff auf die öffentliche Schnittstelle eines Objektes ist in einer objektorientierten Programmiersprache nur durch zwei Konzepte zu verhindern. Erstens kann die Verwendung von öffentlich sichtbaren Klassenbestandteilen per Definition unterbunden und zweitens kann ein Objekt mit einer öffentlichen Schnittstelle dahingehend von einem umhüllenden Objekt gekapselt werden, dass die öffentliche Schnittstelle des eingeschlossenen Objektes nicht sichtbar ist. Die Tragweite der ersten Restriktion, die die Verwendung von öffentlichen Methoden, *Properties* und Feldern verhindert, geht jedoch weit über die Erfordernisse hinaus. Beispielsweise könnten Objekte, die innerhalb eines *Protocol*-Objektes definiert und instantiiert wurden, niemals auf Methoden und weitere Klassenbestandteile dieser Instanz zugreifen.

Die Bereitstellung einer geeigneten *Protocol*-Hülle, die die öffentliche Schnittstelle der beinhaltenden Instanz kapselt, zeigt daher eine deutliche höhere Eignung.

Die Erweiterung der Programmiersprache *C#* ermöglicht analog zur Einführung des Schlüsselwortes *message* die Einführung des Bezeichners *protocol* und ermöglicht damit wiederum die automatische Generierung von *protocol*-Quellcode. Auflistung 3.39 zeigt die Verwendung des neu eingeführten Schlüsselwortes durch die Definition der minimalistischen *Protocol*-Klasse *MyProtocol*.

```
1       public protocol MyProtocol
2       {
3           protected void Execute()
4           {
5           }
6       }
```

Auflistung 3.39: Verwendung des Schlüsselwortes protocol

Das in Auflistung 3.40 dargestellte Grundgerüst wird von der *C#*-Compilererweiterung generiert, sobald der Compiler bei der Übersetzung auf eine mit dem Schlüsselwort *protocol* gekennzeichnete Klasse trifft. Das Gerüst besteht aus der *Wrapper*-Klasse *PublicProtocol* sowie der Klasse *MyProtocol*, die über das *Factory-Pattern* mittels der *New*-Methode instantiiert werden kann. Die erzeugte Struktur stellt sicher, dass mit Ausnahme der *Wrapper*-Klasse kein Objekt, das nicht vor der eingeschlossenen *Protocol*-Instanz erzeugt wurde, eine Referenz auf diese halten kann. Um sicherzustellen, dass eine *Protocol*-Instanz keine Referenz auf sich selbst in ein Nachrichtenobjekt integriert, die über die Nachrichtenwarteschlange an weitere *Protocol*-Instanzen verbreitet werden könnte, muss der erweiterte *C#*-Compiler dieses abprüfen.

Die zweite Herausforderung für die Sicherstellung der ausschließlich Nachrichtenbasierten Kommunikation bieten die statischen Bestandteile innerhalb der *Protocol*-Klassen, da diese per Definition geteilten Speicherraum für alle *Protocol*-Instanzen eines Prozessraumes darstellen. Die Definition statischer Objekte birgt neben dem Faktum des geteilten Speichers den Nachteil, dass die Verteilung von *Protocol*-Instanzen über Prozessgrenzen hinaus beeinträchtigt wird. Es ist daher anzuraten, gemeinsam genutzte Informationen den interessierten *Protocol*-Instanzen per Nachricht zuzustellen und die Definition der Reichweite eines statischen Objektes von Prozess- auf *Protocol*-weit zu beschränken.

Sobald eine *Protocol*-Instanz Kindobjekte beinhaltet, die auf einen geteilten Datenspeicher zugreifen möchten, sollten diese Daten im gemeinsamen Vater-*Protocol* abgelegt werden. Der gemeinsame Zugriff auf die Variable einer *Protocol*-Instanz ist aufgrund der vom *Scheduler* kontrollierten Ausführungsumgebung sicher. Damit ein

```
1       public class PublicProtocol
2       {
3           private ProtocolBase protocol;
4           public PublicProtocol(ProtocolBase protocol)
5           {
6               this.protocol = protocol;
7           }
8       }
9
10      public class MyProtocol : ProtocolBase
11      {
12          public static PublicProtocol New()
13          {
14              return new PublicProtocol(new MyProtocol());
15          }
16
17          private MyProtocol()
18          {
19          }
20
21          protected override IEnumerable<ReceiverBase> Execute()
22          {
23          }
24      }
```

Auflistung 3.40: Generierter protocol Quellcode

Kindobjekt jedoch auf den gemeinsamen Datenspeicher zugreifen kann, muss dieses über eine Referenz auf die in der Objekthierarchie höher stehende *Protocol*-Instanz verfügen. Das immerwährende Mitführen der Instanzreferenz in einer beliebig tief verschachtelten Objekthierarchie ist jedoch sehr implementierungsaufwändig und kostet wertvolle Entwicklungszeit.

Die Bereitstellung des erweiterten *C#*-Compilers, der die Semantik des Schlüsselwortes *static* von Prozessraum auf *Protocol*-Raum einschränkt, erlaubt es, von einem beliebigen Kindobjekt einer *Protocol*-Instanz eine statische Methode beziehungsweise ein statisches *Property* auf der *Protocol*-Klasse aufzurufen. Der Aufruf des statischen Konstruktes wird dann auf die aktuell durch den *Scheduler* ausgeführte *Protocol*-Instanz weitergeleitet und bietet somit den einfachen Zugriff auf die zu einem Kindobjekt zugehörige *Protocol*-Instanz.

Die *C#*-Compiler beziehungsweise dessen Erweiterung wandelt dazu jedes statische Objekt innerhalb einer *Protocol*-Klasse um und instantiiert ein eigenes Objekt für jede erstellte Instanz. Jeder *Protocol*-Instanz steht damit eigener, ungeteilter Speicher zur Verfügung, der von den Kindobjekten genutzt werden kann. Die Ermittlung der aktuell ausgeführten *Protocol*-Instanz erfolgt über lokalen Speicher des ausführendes

Threads. Die *Scheduler* des *Gears4Net*-Programmiermodells notieren die Referenz der jeweils ausgeführten *Protocol*-Instanz im als *Thread local Data (TLD)* [29] beziehungsweise *Thread local Storage (TLS)* [82, 88] bezeichneten Datenspeicher. Sobald ein Kindobjekt eine statische Methode oder ein statisches *Property* aufruft, wird die aktuell ausgeführte *Protocol*-Instanz über eine Abfrage der *TLD* ermittelt. Der statische Zugriff kann somit direkt an die aktuell ausgeführte Instanz weitergeleitet werden und minimiert damit den manuellen Entwicklungsaufwand.

3.9.3 Verhinderung blockierender Systemaufrufe

Den Abschluss der Weiterentwicklung findet das *Gears4Net*-Programmiermodell mit der Verhinderung des Aufrufs von blockierenden Systemfunktionen und nativen *Thread*-Bibliotheken. Diese Funktionen und Bibliotheken werden automatisiert von der *C#*-Sprach- und Compilererweiterung erkannt und führen instantan zu Compilerfehlern. Die Entwickler werden durch diese Maßnahme zur Verwendung nichtblockierender Aufrufe angehalten.

3.10 Verwandte Arbeiten

Die Evolution der *Multi-Core*-Prozessorarchitekturen hat die Entwicklung von parallelen Programmiermodellen signifikant voranschreiten lassen. Trotz unterschiedlicher Zielstellung wurde das *Gears4Net*-Programmiermodell von einigen Modellen inspiriert, deren Entwicklung seit den 70er Jahren vorangetrieben wurde. Das folgende Kapitel betrachtet verschiedene Arbeiten und Veröffentlichungen des Themenbereiches und grenzt diese von der Konzeption und Implementierung des *Gears4Net*-Programmiermodells ab. Dabei werden sowohl neue Programmiersprachen sowie Klassenbibliotheken und einfache Synchronisierungsparadigmen betrachtet.

3.10.1 Comega und Polyphonic C#

Der Abschnitt „Verwandte Arbeiten“ beginnt mit der Vorstellung der Programmiersprache *Polyphonic C#* [12, 13], die einen wesentlichen Bestandteil der Spracherweiterung *Comega* [15] bildet. Die Spracherweiterung wurde von Microsoft Research vorgestellt und ergänzt die *ECMA*-standardisierte Programmiersprache C# [38] um asynchrone Methodenaufrufe und Synchronisierungsmechanismen und vereinfacht

somit die Ausdrucksweise von paralleler Verarbeitung auf der Ebene der Programmiersprache.

Asynchrone Methoden

Die Einführung der asynchronen Methodenaufrufe geht mit der Bereitstellung des Schlüsselwortes *async* einher. Methoden, die mit diesem Schlüsselwort gekennzeichnet sind, unterscheiden sich signifikant von standardkonformen blockierenden Aufrufen, da diese Methoden direkt nach dem Aufruf zurückkehren und die Rümpfe dieser Methoden asynchron, beispielsweise von einem anderen *Thread*, ausgeführt werden. Die mit dem Schlüsselwort *async* gekennzeichneten Methoden unterliegen zudem der Einschränkung, dass sie über keinen Rückgabetyp verfügen und innerhalb des Methodenrumpfes keine *Exceptions* ausgelöst werden dürfen. Der Verzicht auf einen Rückgabetyp erklärt sich durch die asynchrone Ausführung, da der Methodenrumpf in der Regel noch nicht vollständig ausgeführt wurde, wenn die Methode zurückgekehrt ist.

```
1   public class Buffer
2   {
3       public async Put(string s)
4       {
5           // body will be executed asynchronously
6       }
7
8       public string Get() & public async Put(string s)
9       {
10          return s;
11      }
12  }
13
14  public class Program
15  {
16      static void Main()
17      {
18          Buffer b = new Buffer();
19          b.Put("blue");
20          b.Put("sky");
21          Console.WrtiteLine(b.Get() + b.Get());
22      }
23  }
```

Auflistung 3.41: Asynchroner Methodenaufruf [13]

Die Signatur einer asynchronen Methode ist am Beispiel eines *Buffers* dokumentiert, zu dem Daten über die *Put*-Methode hinzugefügt beziehungsweise über die *Get*-Methode entnommen werden können. Der zugehörige Quelltext der *Buffer*-Klasse

mit den beiden vorgestellten Methoden ist in Auflistung 3.41 dargestellt und in den folgenden Abschnitten erläutert.

Synchronisierungsmechanismen

Die Synchronisierungsmechanismen der Sprache Polyphonic C# werden als *Chords* beziehungsweise *Join Pattern* bezeichnet. Sie bestehen aus Kopf und Rumpf, wobei der Kopf eine Menge an Methodendeklarationen enthält, während der Rumpf Funktionalität beinhaltet, die ausschließlich dann ausgeführt wird, wenn alle im Kopf deklarierten Methoden ausgeführt wurden. Die Konkatenation der Methodendeklarationen im Kopf eines *Chord* erfolgt über das Zeichen des logischen Und-Operators &, wie dieses am Beispiel des *Chords* in Zeile 9 des Beispiels dargestellt ist, der aus den Methoden *Get* und *Put* besteht.

Die der Sprache Polyphonic C# zugrundeliegende Ausführungsumgebung verwaltet die Ausführung der verschiedenen Methoden in Warteschlangen und erkennt anhand dieses Verfahrens, ob ein im Quellcode definierter *Chord* erfüllt ist. Das Ausführungsverhalten der beiden Methoden *Get* und *Put* wird nach [13] wie folgt definiert:

- Sollte die synchrone Methode *Get* aufgerufen werden und es befindet sich ein ausgeführter asynchroner *Put*-Aufruf in der Warteschlange, der noch keinem *Chord* zugewiesen ist, so wird die *Put*-Methode der Warteschlange entnommen, und der *Chord* kann direkt zu Ende laufen.

- Befindet sich jedoch zum Zeitpunkt des synchronen Aufrufs der Methode *Get* kein ausgeführter asynchroner *Put*-Aufruf in der Warteschlange, so blockiert der ausführende *Thread* solange, bis ein passender *Put*-Aufruf getätigt wurde.

Neben dem beschriebenen Ausführungsverhalten wird auch das *Thread-Scheduling* von der Ausführungsumgebung organisiert. Enthält ein *Chord* mindestens eine nicht synchrone Methode, wie dies im Beispiel in Auflistung 3.41 dargestellt ist, so wird die synchrone Methode im Kontext des die Methode aufrufenden *Threads* ausgeführt. Andernfalls werden die asynchronen Methoden in neu erzeugten *Threads* beziehungsweise durch allokierbare *Threads* aus dem *Thread-Pool* ausgeführt.

Das nicht-deterministische Ausführungsverhalten der *Chords* wird an einem einfachen Beispiel deutlich, das ab Zeile 18 dargestellt wird. Die beiden Werte *blue* und *sky* werden jeweils durch den Aufruf der Methode *Put* auf einer Instanz der *Buffer*-Klasse zur internen Warteschlange hinzugefügt und anschließend im Zuge einer

Ausgabefunktion abgerufen. Die Ausgabefunktion, die sich zweier Aufrufe der *Get*-Methode bemächtigt, gibt entweder die Zeilenkette *blue sky* oder *sky blue* aus. Die Reihenfolge dieser Ausgabe ist nicht-deterministisch und abhängig vom *Scheduling*, das bei der Ausführung der asynchronen *Put*-Methoden Anwendung findet.

Abgrenzung

Polyphonic C# bildet die Grundlage für die asynchrone und parallele Programmierung in der Spracherweiterung *Comega*. Das vorgestellte Konzept erlaubt den sehr feingranularen Ausdruck von Parallelität auf Sprachebene und ermöglicht dabei, einzelne Methoden oder Methodenketten parallel auszuführen und zu synchronisieren. *Polyphonic C#* unterscheidet sich bereits daher konzeptionell wesentlich vom Programmiermodell *Gears4Net*, das Parallelität als Designkriterium betrachtet und zugleich ein intuitives Programmierkonzept bietet. Die unterschiedlichen Konzepte führen sich in den jeweiligen Ausführungsumgebungen fort. Während das *Gears4Net*-Programmiermodell *Protocol*-Instanzen und Zustandsmaschinen als abgeschlossene Quellcodeblöcke betrachtet, die pseudo-parallel und damit synchronisierungsfrei ausgeführt werden können, setzt *Polyphonic C#* weiterhin auf echte Parallelität und damit auf die Problematik der Synchronisierung. Die Ausnutzung der Mehrprozessorarchitekturen erfolgt beim *Gears4Net*-Modell somit nicht durch die parallele Ausführung von einzelnen Methoden, sondern von unabhängigen *Protocol*-Instanzen, die ausschließlich über Nachrichten miteinander kommunizieren und daher keinerlei Synchronisierung bedürfen.

3.10.2 Concurrency and Coordination Runtime

Die Microsoft *Concurrency and Coordination Runtime (CCR)* [57, 76, 100, 115] ist ein Nachrichten-basiertes Programmiermodell für die parallele und verteilte Anwendungsentwicklung. Das Programmiermodell der *CCR*, das durch den *Pi-Calculus* [84, 108] inspiriert wurde, stellt *Ports* bereit, die Nachrichtenobjekte aufnehmen und an geeignete *Receiver* weiterleiten können. Eine *Port*-Instanz kann beliebig viele Nachrichten aufnehmen, wobei die Datentypen, die ein *Port* aufnehmen kann eingeschränkt werden können. Je nach Konfiguration kann ein *Port* bis zu 16 unterschiedliche Nachrichtentypen verwalten.

Die *CCR* wurde nicht als Spracherweiterung wie das im vorherigen Abschnitt vorgestellte Modell *Polyphonic C#* sondern als Bibliothek implementiert und kann somit von jeder Microsoft .NET Applikation referenziert werden. Die Bibliothek baut

jedoch auf den in der Sprache *Polyphonic C#* vorgestellten Synchronisierungsmechanismen auf und bildet diese in Form von *Ports* und *Receivern* ab.

Darüber hinaus ist die *CCR* das erste bekannte Programmiermodell, das die Verwendung der Iteratoren als unterbrechbare Methoden vorstellt. Sie bildet in diesem Punkt die Grundlage für die Entwicklung des *Gears4Net*-Programmiermodells.

Funktionsweise der Port- und Arbiter-Instanzen

Ein *Port* ist eine Klasse, die in verschiedenen Parametrisierungen instantiiert werden kann. Die Spezifikation der Datentypen, die von einem *Port* aufgenommen werden können, erfolgt über das Konzept der generischen Datentypen (siehe Kapitel 2.1). Die Anzahl der anzugebenden Datentypen ist auf 16 Einträge limitiert. Die Limitierung ist willkürlich gewählt, begründet sich aber in dem Faktum, dass die Verwendung der generischen Schreibweise für jede der 16 möglichen Konfigurationen die Bereitstellung einer eigenen von *Port* erbenden Klasse erfordert. Die technische Realisierung der *Ports*, die mehr als einen Datentyp enthalten, ist hingegen sehr skalierbar. Ein *Port* verfügt über eine interne Datenstruktur aus typisierten *Ports*, die genau einen Datentyp aufnehmen können. Verwaltet ein *Port* beispielsweise 16 Datentypen, so verfügt dieser über eine Liste mit 16 *Port*-Instanzen, die jeweils genau einen Datentyp beherbergen.

Die Nachrichten, die in einen *Port* eingestellt werden, können über die Spezifikation von Wartebedingungen innerhalb eines Iterators entgegengenommen werden. Die *Port*-Instanz stellt dazu die Methode *with* bereit, der ein Funktionszeiger auf eine Methode übergeben wird, die die ankommende Nachricht weiterverarbeiten kann. An eine *Port*-Instanz können über den wiederholten Aufruf der *with*-Methode verschiedenste Funktionszeiger übergeben werden. Dies ist besonders wichtig, wenn ein *Port* mehr als nur einen Datentyp verwaltet. Über dieses Verfahren können beispielsweise zwei Funktionszeiger auf eine den Datentyp *int* und eine den Datentyp *string* verarbeitende Methode übergeben werden. Der *Port* entscheidet abhängig von der Signatur des übergebenen *Delegates*, welcher Methode die ankommende Nachricht übergeben wird. Im Falle zweier Funktionszeiger mit identischer Signatur entscheidet die *Port*-Instanz in einem nicht-deterministischen Verfahren, welcher der beiden Methoden die zu verarbeitende Nachricht zugestellt wird, wobei das Verfahren in keinerlei Weise vom Entwickler beeinflusst werden kann.

Zur Darstellung von komplexeren Wartebedingungen führt die *CCR*-Bibliothek, den *Choice-Arbiter* für disjunkte Wartebedingungen sowie den *Join-Arbiter* für die konjunktive Variante ein. Die Verwendung des *Choice-Arbiter*-Konzeptes ist in Auf-

```
1       activate(p.with(MyIntHandler) | p.with(MyStringHandler));
```

Auflistung 3.42: Choice Arbiter [115]

listung 3.42 dargestellt, die des *Join-Arbiter* folgt im Anschluss in Auflistung 3.43. Der beispielhaft implementierte *Choice-Arbiter* definiert eine Bedingung, die auf den Eingang einer Nachricht der Typen *int* beziehungsweise *string* wartet. Aus der Darstellung des Quelltextes ist nicht erkennbar, auf welchen Nachrichtentyp mittels des Aufrufs der *with*-Methode gewartet wird. Lediglich die Bezeichnung der Methode, auf die der *Delegate* zeigt, gibt Aufschluss über den definierten Datentyp.

```
1       join<int,string>(p1, p2).with(Handler);
```

Auflistung 3.43: Join Arbiter [115]

Der *Join-Arbiter* agiert analog zur präsentierten *Choice*-Variante mit dem Unterschied, dass dieser erst feuert, sobald alle spezifizierten Nachrichten eingetroffen sind. Die Quellcodedarstellung zeigt zudem, dass das *Join*-Kommando die erwarteten Datentypen in generischer Parameterdeklaration spezifiziert und die *Port*-Instanzen, auf denen die Daten eintreffen, als Parameter in die *join*-Methode hineingibt. Abgeschlossen wird der Aufruf eines *join*-Kommandos mit der Angabe des Funktionszeigers, der im Anschluss an das Eintreffen des Gesamtereignisses ausgeführt wird.

Abgrenzung

Das *Gears4Net*-Programmiermodell adaptiert das Iteratorenkonzept der *CCR*-Klassenbibliothek und weist daher naturgemäß Ähnlichkeiten auf. *Gears4Net* differenziert sich jedoch deutlich im Konzept der Ausführungsumgebung durch die Einführung des Konzeptes der *Scheduler*, die für die Verwaltung von *Protocol*-Instanzen und den darin enthaltenen *State-Machines* zuständig sind. Im Gegensatz zur *CCR* führt das *Gears4Net*-Programmiermodell die *Protocol*-Instanzen pseudoparallel aus und kann daher vollständig ohne Synchronisierungsmechanismen auskommen. Eine weitere Differenzierung erfolgt über die in *Gears4Net* eingeführte *Broadcast-Message-Queues*, deren Semantik sich deutlich vom vorgestellten *Port*-Konzept abhebt, und der Verwendung von *Signal*-Objekten, die ebenfalls nicht im Einklang mit dem Kommunikationsmodell der *CCR* steht.

3.10.3 Parallel Extensions

Neben der Programmiersprache *Polyphonic C#* und der Klassenbibliothek *CCR* bilden die *Parallel Extensions* [80, 120] das dritte und letzte von Microsoft vorgestellte Programmiermodell für parallele Anwendungen. Die *Parallel Extensions* werden mit der Einführung von Microsoft .NET 4.0 beziehungsweise Visual Studio 2010 ausgeliefert und integrieren sich als Bibliothek in den Namensraum *System.Concurrency*. Die Bibliothek setzt sich aus drei Bestandteilen zusammen. Den ersten Block bildet die *Task Parallel Library (TPL)* [81], die die Unterstützung für die Parallelisierung von Imperativen Ausdrücken ermöglicht. Der zweite Bestandteil der Bibliothek erweitert die bestehende Funktionalität der *Language Integrated Query (LINQ)* [59] und bietet mit *Parallel LINQ (PLINQ)* [72, 79] die Möglichkeit paralleler Ausführung mittels deklarativer Spezifikation. Abgeschlossen werden die Bestandteile der *Parallel Extensions* durch die Einführung von synchronisierten Datenstrukturen, den *Coordination Data Structures (CDS)* [78].

```
1   void MatrixMult(int size, double[,] m1, double[,] m2, double[,] result)
2   {
3       Parallel.For( 0, size, delegate(int i)
4       {
5           for (int j = 0; j < size; j++)
6           {
7               result[i, j] = 0;
8               for (int k = 0; k < size; k++)
9               {
10                  result[i, j] += m1[i, k] * m2[k, j];
11              }
12          }
13      });
14  }
```

Auflistung 3.44: Matrixmultiplikation mit Parallel.For [32]

Die Funktionsweise der *Parallel Extensions* wird in Auflistung 3.44 am Beispiel einer teilweise parallelisierten Matrixmultiplikation mit Hilfe der *Parallel.For*-Schleife dargestellt. Im Gegensatz zur *for*-Schleife, die nativ in die Programmiersprache *C#* integriert wurde, ist die *Parallel.For*-Schleife als statische Methode innerhalb der *Parallel*-Klasse deklariert. Die Methode erwartet die Limitierungen des Schleifendurchlaufs in den ersten beiden Parametern. Im Fall des angegebenen Beispiels läuft die Schleife von *0* bis *size*. Der dritte Parameter der *For*-Schleifemethode erwartet einen Funktionszeiger auf eine Methode, die bei jedem Schleifendurchlauf ausgeführt werden soll. Die Ausführung dieses Methodenrumpfes erfolgt nun im Gegensatz zu einer normalen *For*-Schleife in paralleler Ausführung.

Zur Erhaltung der charakteristischen Darstellung einer *For*-Schleife wird der dritte Parameter der *Parallel.For*-Methode nicht als expliziter Zeiger auf eine Methode, sondern als anonyme Methode [112] übergeben. Diese beginnt mit der Angabe des Schlüsselwortes *delegate* in Zeile 3 und endet in Zeile 13.

Abgrenzung

Die *Parallel Extensions* bildet mit ihren drei Bestandteilen *TPL*, *PLINQ* und *CDS* ein sehr elegantes Modell zur Parallelisierung von imperativen und deklarativen Sprachkonstrukten und stellt des weiteren eine Menge synchronisierter Datenstrukturen bereit. Die Klassenbibliothek der *Parallel Extensions* kapseln die für die Parallelisierung der Schleifendurchläufe notwendigen *Thread*-Operationen und nehmen dem Nutzer der Bibliothek das Erzeugen und Synchronisieren von *Threads* ab.

Das Programmiermodell der *Parallel Extensions* dient dem Zwecke der Parallelisierung von unabhängigen, rechenintensiven Schleifendurchläufen und deklarativen Ausdrücken. Im Gegensatz zum Programmiermodell *Gears4Net* versteht es sich jedoch nicht als ein Modell, dessen Ziel ein vollständig paralleles Anwendungsdesign ist. Die beiden Modelle unterscheiden sich daher signifikant in der Granularität der Parallelisierung.

3.10.4 Parallel C#

Die Programmiersprache *Parallel C#* [46] ist eine Erweiterung der Sprache *C#*, die durch die Sprachen *Polyphonic C#* [12, 13], *T++* [2, 89, 90] und die funktionale Variante *ML* [83, 85] inspiriert wurden. Die Erweiterung beinhaltet *Join-Pattern*, asynchrone Methoden sowie Funktionen, die als *movable* gekennzeichnet sind und auf Basis des *Distributed Runtime System (DRS)* parallel in einem *Cluster*, einem *Meta-Cluster* beziehungsweise einem *GRID* ausgeführt werden können.

Das in die Sprache *Parallel C#* eingeführte Schlüsselwort *movable* kennzeichnet eine Methode, die auf einem anderen Rechner ausgeführt werden soll in ähnlicher Weise, wie das Schlüsselwort *async* die asynchron auszuführenden Methoden der Sprache *Polyphonic C#* charakterisiert. Der Aufruf einer *movable*-Methode unterliegt derselben Einschränkung, wie dies beim Aufruf einer asynchronen Methode der Fall ist, und bietet aufgrund des asynchronen Verhaltens keine Möglichkeit der Angabe eines Rückgabetyps. Das Aufrufverhalten der beiden Varianten unterscheidet sich jedoch signifikant, da die als *movable* gekennzeichneten Methoden intern über die Indirektion eines *Proxies* angesprochen werden. Sobald eine solche Methode aufgerufen wird,

werden alle übergebenen Parameter serialisiert und auf den entsprechenden Endpunkt übermittelt. Im Fall des Aufrufs der Methode auf einem Objekt und nicht auf einer statischen Klasse wird der Aufrufer ebenfalls dem Serialisierungsverfahren unterworfen. Insofern der Aufruf der *movable*-Methode den Status des Objektes verändert, ist zu beachten, dass die durchgeführte Modifikation auf der serialisierten Kopie und nicht auf dem aufrufenden Objekt vollzogen wird.

Abgrenzung

Die vorgestellte Programmiersprache *Parallel C#* bietet interessante Erweiterungen für *Cluster* und *GRID* basierte Anwendungen. Aufgrund der deutlichen Verwandtschaft zur Sprache *Polyphonic C#* grenzt sich *Parallel C#* in ähnlicher Weise vom *Gears4Net*-Programmiermodell ab. Beide Varianten betrachten Parallelität auf einer ähnlich feingranularen Ebene und nicht als generelles Applikationsdesignziel, wie dies durch das Programmiermodell *Gears4Net* propagiert wird.

3.10.5 Scala

Die Programmiersprache *Scala* [47, 48] wurde im Jahre 2001 unter der Federführung von Odersky an der École Polytechnique Fédérale de Lausanne (EPFL) mit der Zielsetzung entwickelt, die Komponentenorientierung und die damit verbundene Wiederverwendbarkeit zu stärken. Die statisch typisierte Programmiersprache vereinigt die Konzepte der objektorientierten mit denen der funktionalen Programmierung. Die resultierende Syntax ist dabei an die Sprachen *Java* [93] und *ML* [83, 85] angelehnt, die aus dem objektorientierten beziehungsweise funktionalen Umfeld entspringen.

Die Programmiersprache besticht durch die Vereinheitlichung und Generalisierung der beiden Programmierkonzepte und erweitert diese um Funktionalitäten der Abstraktion, Komposition und Dekomposition von Komponenten. Der Gedanke der Wiederverwendbarkeit und der Komponentenorientierung spiegelt sich bei der Programmiersprache *Scala* durch die Einbindung moderner virtueller Maschinen und deren Bibliotheken wider. *Scala* ist daher sowohl auf der *Java Virtual Machine (JVM)* [93] als auch auf der Microsoft *Common Language Runtime (CLR)* [39, 43] ausführbar und ermöglicht somit den Zugriff auf bestehende Bibliotheken und bereits existierende Softwarekomponenten.

Der Programmiersprache *Scala* stehen neben den Bibliotheken der *JVM* beziehungsweise der *CLR* auch eigene Pakete, wie beispielsweise die *scala.actors* Bibliothek,

die das Aktorenmodell implementiert, zur Verfügung. Eine exemplarische Implementierung des *Ping-Pong*-Protokolls, bei dem auf jede *Ping*-Nachricht mit einer *Pong*-Nachricht geantwortet wird, ist in Auflistung 3.45 dargestellt. Das Verfahren terminiert nach einer definierten Anzahl an versandten *Ping*-Nachrichten und signalisiert dies der Gegenstelle durch den Versand einer *Stop*-Nachricht.

Das präsentierte Beispiel zeigt die statische Klasse *Ping*, die von der Klasse *Actor* aus dem Paket *scale.actors* erbt und über einen Konstruktor mit den zwei Parametern *count* und *pong* des Typs *int* beziehungsweise *Actor* verfügt. Der Konstruktor erlaubt über diese Parameter die Konfiguration der Anzahl der zu versendenden *Ping*-Nachrichten sowie die Angabe der Gegenstelle, der die Nachrichten zugestellt werden sollen. Die Implementierung der Methode, innerhalb der die Applikationslogik abgelegt ist, erfolgt ab Zeile 3 mit der Deklaration der parameterlosen Funktion *act*. Der Versand der ersten *Ping*-Nachricht an die *Pong*-Gegenstelle erfolgt im Anschluss an die Dekrementierung der *pingsLeft*-Variablen durch die Verwendung des *!*-Operators in Zeile 6. Die folgende *while*-Schleife wird solange durchlaufen, bis die definierte Anzahl *Ping*-Nachrichten versandt und die Abbruchbedingung in Zeile 20 erreicht ist.

Der Nachrichtenempfang des *Actors* wird durch das Kommando *receive* in Zeile 9 initialisiert und mittels des *case*-Kommandos genauer spezifiziert. Der *Actor* wartet im gegebenen Beispiel blockierend auf den Eingang einer *Pong*- respektive *Stop*-Nachricht. Abhängig vom Typ der eingehenden Nachricht wird entweder der *case*-Ausdruck in Zeile 11 oder der zur Terminierung führende Ausdruck in Zeile 22 ausgeführt. Gesetz des Eingangs der *Pong*-Nachricht antwortet der *Ping*-Aktor mit einer *Ping*-Nachricht, sofern das Limit der versandten Nachrichten noch nicht erreicht wurde.

```
1   class Ping(count: int, pong: Actor) extends Actor
2   {
3       def act()
4       {
5           var pingsLeft = count - 1
6           pong ! Ping
7           while (true)
8           {
9               receive
10              {
11                  case Pong =>
12                      if (pingsLeft > 0)
13                      {
14                          pong ! Ping
15                          pingsLeft -= 1
16                      }
17                      else
18                      {
19                          pong ! Stop
20                          exit()
21                      }
22                  case Stop =>
23                      exit()
24              }
25          }
26      }
27  }
```

Auflistung 3.45: Ping-Pong Szenario auf Basis des Aktorenmodells der Sprache Scala [47]

Abgrenzung

Die vorgestellte Programmiersprache *Scala* bietet neben der Verbindung von objektorientierter und funktionaler Konzepte reichhaltige Bibliotheken, die unter anderem das im *Gears4Net*-Programmiermodell implementierte Aktorenmodell bereitstellen. Die Implementierung des Aktorenmodells unterscheidet sich jedoch signifikant in den Punkten des Programmiermodells, der *Scheduling*-Verfahren, der *State-Machines*, der Komplexität der Wartebedingungen sowie der Flusskontrolle.

Die Programmiersprache *Scala* erlaubt die Verwendung von blockierenden und nicht-blockierenden Systemaufrufen und Wartebedingungen. Das vorgestellte Beispiel in Auflistung 3.45 verwendet das blockierende *receive*-Kommando, das den ausführenden *Thread* solange schlafen legt, bis eine Nachricht eingegangen ist, die der Wartebedingung genügt. Die Nachteile des blockierenden Verfahrens können jedoch durch die Verwendung des nicht-blockierenden Kommandos *react* aufgehoben werden. Es ist jedoch anzumerken, dass der Einsatz der *react*-Anweisung weiteren Einschränkungen unterliegt. Beispielsweise ist der Einsatz des Kommandos innerhalb eines Schleifenrumpfes untersagt.

Die Verantwortung bezüglich der *Scheduling*-Verfahren, die die *Actor*-Instanzen ausführen, liegt in der *Runtime* der Programmiersprache *Scala* und deren Bibliotheken. Die *Actor*-Instanzen werden standardmäßig von vier *Threads* aus einem *Pool* ausgeführt. Sollten alle *Threads* blockieren, werden automatisch weitere *Threads* erzeugt, die die weitere Ausführung der *Actor*-Instanzen garantieren. Die *Scheduling*-Mechanismen unterscheiden sich deutlich von denen des *Gears4Net*-Modells, bei dem je nach Anwendungszweck spezialisierte *Scheduling*-Algorithmen eingesetzt werden können. Beispielweise ermöglicht die Verwendung des *WinFormsSchedulers* (siehe Kapitel 3.6.6.2) die Ausführung beliebig vieler *Protocol*-Instanzen innerhalb nur eines *Threads*.

Das *Gears4Net*-Programmiermodell erlaubt die Aufspaltung eines komplexen Automaten in eine beliebige Menge an Zustandsmaschinen, die *Race Condition*-frei durch den gleichen *Scheduler* ausgeführt werden. Die Implementierung solcher Teilautomaten ist im Aktorenmodell der Sprache *Scala* nicht vorgesehen. Die Aufspaltung der Funktionalität auf mehrere Funktionen oder *Actor*-Instanzen würde die Ausführung im Kontext eines identischen *Threads* nicht sicherstellen und somit mögliche *Race Conditions* verursachen.

Die Darstellung komplexer Wartebedingungen bildet den vierten Kritik- beziehungsweise Differenzierungspunkt der beiden Implementierungen des Aktorenmodells. Das *Gear4Net*-Programmiermodell ermöglicht die kompakte Darstellung komplexer Wartebedingen durch die Verwendung der logischen Operatoren *UND* und *ODER* und unterscheidet sich damit tief gehend von der Darstellungsweise der Programmiersprache *Scala*. Diese implementiert ein logisches *ODER* durch die Angabe mehrerer *case*-Anweisungen innerhalb eines *receive*-Blocks, wie dieses in den Zeilen 11 und 22 des Beispiels in Auflistung 3.45 dargestellt ist und erlaubt damit keine kompakte Darstellung. Die Implementierung des logischen *UND* verkompliziert die Spezifikation der Wartebedingung abermals, da dieses ausschließlich durch die Verschachtelung von *receive*-Anweisungen ermöglicht wird. Aufgrund der Tatsache, dass die *receive*-Anweisungen sequentiell abgearbeitet werden, würde eine Wartebedingung, die auf das Eintreten einer Nachricht des Typs A und einer Nachricht des Typs B wartet, zweifach implementiert werden müssen, da das *Scala-Framework* die Reihenfolge des Nachrichteneingangs unterscheidet. Der äußere *receive*-Block müsste somit zwei *case*-Anweisungen enthalten, die jeweils einen weiteren *receive*-Block enthalten und somit die Wartebedingung $A \,\&\, B$ sowie $B \,\&A$ implementieren.

Den Abschluss der Differenzierung bildet die der Nachrichtenübertragung zugrundeliegende Warteschlangentheorie. Während das *Scala*-Modell über eine einfache

Nachrichtenwarteschlange verfügt, die die Nachricht von einem Sender einer empfangenden Wartebedingung zustellt, bietet das *Gears4Net*-Programmiermodell den Einsatz der *BroadcastQueue*. Diese erlaubt die Zustellung an mehrere *Receiver* und stellt zudem Mechanismen der Flusskontrolle bereit, um möglichen Überlastungen entgegenzutreten.

3.10.6 SEDA

Das Modell der *Staged Event-Driven Architecure (SEDA)* [128, 129] wurde von Matt Welsh an der University of California mit dem Ziel konzipiert, die Entwicklung von hoch parallelen Serverapplikationen zu vereinfachen. Das Designmodell ist für die Ausführung von Serverapplikationen wie beispielsweise Web-Applikationen oder Web-Services gedacht, die trotz eines massiv parallelen Nutzungsverhaltens einen hohen Durchsatz gewährleisten müssen, um die erwartete Servicequalität zu erreichen.

Das *SEDA*-Designkonzept, das sich selbst als Betriebssystem für Internet-Dienste versteht, basiert auf der Idee, die gesamte Applikationslogik eines Dienstes in unabhängige *Stages* zu unterteilen, die über *Queues* miteinander kommunizieren. Jede *Stage* verfügt dabei über eine eigene Warteschlange, in die *Tasks* eingestellt werden, die von der jeweiligen *Stage* bearbeitet und an die entsprechend nächste *State* weitergereicht werden. Das Modell der unabhängigen *Stages* ermöglicht eine sehr feingranulare Ressourcenzuweisung. So kann beispielsweise die Rechenleistung für eine sehr lastintensive Berechnung durch ein geeignetes *Scheduling* der entsprechenden *Stage* angepasst werden.

Das Entwicklungsmodell der einzelnen *Stages* basiert auf der Idee der *Event Driven Architecture* [24, 40] und vermeidet etwaige Skalierungsprobleme, die durch den massiven Einsatz von *Threads* entstehen können. Die Warteschlangen beziehungsweise *Task*- oder *Message-Queues* verfügen zudem aus Gründen der Flusskontrolle über Filter- und Sortiermechanismen, die beispielsweise das Hinzufügen von neuen *Tasks* unter Volllast des Systems verhindern.

Die *Staged Event-Driven Architecure* benennt die vier Elementarbestandteile *Stages*, *Tasks*, *Thread Pools* und *Queues*, die die Grundlage für das Applikationsdesign sowie für die Ausführungsumgebung des Modells bilden und in den folgenden Abschnitten vorgestellt werden.

Stage

Das Designmuster des *SEDA*-Konzeptes zerlegt die Applikationslogik sowie den damit verbundenen Datenfluss innerhalb der Applikation in unabhängige Teilkomponenten, die als *Stages* bezeichnet werden. Jede *Stage* wird von einem *Thread* des *Thread Pools* ausgeführt und verarbeitet dabei *Tasks*, die in die *Incoming-Queue* der *Stage* eingestellt werden. Nachdem ein *Task* verarbeitet worden ist, wird dieser in die Warteschlange der nächsten *Stage* eingestellt. Das Verfahren terminiert, sobald die letzte *Stage* im gerichteten Applikationsgraphen durchlaufen wurde.

Die Vorteile dieses *Stage*-basierten Ansatzes sind vielfältig. Die Gliederung in einzelne *Stages* reduziert die Menge der zu implementierenden Zustände und ermöglicht zudem eine logische Gruppierung. Diese Eigenschaft ermöglicht die unabhängige Entwicklung und Wartung sowie das unabhängige Testen der einzelnen Teilkomponenten. Zudem erlaubt die Weitergabe der verarbeiteten *Task*-Objekte an die jeweils nächste *Stage* die Darstellung des Datenflusses in Form eines gerichteten Anwendungsgraphen.

Weitere Stärken des Ansatzes sind im *Scheduling* begründet. Abhängig von der benötigten Last einer *Stage* können sowohl *Scheduling*-Verfahren als auch Replikations- und Verteilungsmechanismen angepasst und konfiguriert werden.

Tasks

Typisierte Nachrichten, die eine Beschreibung der Arbeitstätigkeit sowie die für die Durchführung der Arbeit relevanten Nutzdaten enthalten, werden im *SEDA*-Designmodell als *Tasks* bezeichnet. Jeder *Task* wird dabei gemäß des Anwendungsgraphen von *Stage* zu *Stage* weitergereicht, bis der Endzustand des Graphen erreicht ist. Die Verarbeitung der *Tasks* kann sowohl sequentiell als auch parallel durchgeführt werden. Im Falle einer parallelen Verarbeitung zieht dies jedoch etwaige Synchronisierungsmechanismen nach sich, da die parallel agierenden *Stages* von verschiedenen *Threads* ausgeführt werden können und somit den Nährboden für *Race Conditions* bilden.

Thread-Pools

Als *Thread-Pool* wird eine Ansammlung von *Threads* bezeichnet, die für die Ausführung einer *SEDA* verwendet werden. Ein *SEDA-Thread-Pool* kann entweder aus autarken *Threads* oder aus *Threads* des Betriebssystems *Thread-Pools* bestehen. Ein in Ausführung befindlicher *Thread* wird dabei mit einer *Stage* beziehungsweise dem aktiven *Task* einer *Stage* assoziiert.

Die Eigenschaften des *Threads-Pools* sind im Bezug auf dessen Größe und die zugrundeliegenden *Scheduling*-Verfahren konfigurierbar, um den maximalen Durchsatz der Gesamtapplikation zu gewährleisten.

Queues

Abgeschlossen wird die Beschreibung der vier Elementarbestandteile durch die Einführung der *Queues*. Diese sind für die Aufnahme der *Task*-Objekte innerhalb jeder *Stage* zuständig. Die *Queues* werden systemintern als *Task*-Listen repräsentiert. Sie erlauben die Angabe von Filterkriterien sowie wie die Reorganisation der Nachrichtenreihenfolge und bilden damit ein effektives Werkzeug zur Datenflusskontrolle.

Abgrenzung

Mit der *Staged Event-Driven Architecure* wurde ein Designmodell zur Implementierung massiv paralleler Serveranwendungen vorgestellt, das eine saubere Softwarearchitektur bei gleichzeitig hohem Durchsatz zulässt. Die Differenzierung der beiden, auf dem Aktorenmodell aufsetzenden Modelle *SEDA* und *Gears4Net* offenbart sich in deren Zielsetzung. Während das *SEDA*-Modell ausschließlich architekturelle Aspekte betrachtet, fokussiert das *Gears4Net*-Modell zusätzlich auf die Art und Weise der Programmierung und verdeutlicht dieses mit dem Zustandsmaschinen-ähnlichen Programmierkonzept, den Strukturierungsmechanismen der *State-Machines* und der intuitiven Darstellbarkeit von höchst komplexen Wartebedingungen.

3.10.7 Pfadausdrücke

Pfadausdrücke (engl. Path Expressions) [20] sind ein Mechanismus zur Beschreibung von Ausführungssequenzen, die beispielsweise ihre Verwendung in nebenläufigen Systemen finden. Campbell und Kolstad [21] zeigen die Anwendung des Konzeptes in der Programmiersprache *Pascal* [68] und erlauben damit beispielsweise Ausdrücke der Form {*4: read: ,write*}, die spezifizieren, dass entweder 4 parallele Lesezugriffe oder genau ein schreibender Zugriff simultan erfolgen darf.

Abgrenzung

Die im *Gears4Net*-Programmiermodell vorgestellten *Receiver*-Wartebedingungen bilden eine Obermenge der ursprünglichen Pfadausdrücke und erweitern das Modell

um die vorgestellten Konzepte und Kommandos des *Gears4Net*-Modells. Der Ausdruck {*4: read: ,write*} lässt sich beispielsweise als *yield return*-Anweisung einer *Gears4Net*-Zustandsmaschine in der Form *yield return Parallel(4, read) | write* definieren. Die Umsetzung der Pfadausdrücke unterscheidet sich zudem von der von Campbell und Kolstad [21] vorgestellten Variante, da das *Gears4Net*-Programmiermodell im Gegensatz zur Implementierung innerhalb der Sprache *Pascal* keinerlei Änderungen an der Programmiersprache selbst vorgenommen hat. Darüber hinaus ermöglicht die *Gears4Net*-Implementierung eine dynamische Auswertung der Ausdrücke zur Laufzeit und ist nicht an die statische Überprüfung des *Compilers* gebunden, wie dies bei der Programmiersprache *Pascal* der Fall ist.

3.10.8 Monitore

Mit dem von Hoare [53, 54] und Brinch Hansen [49] in den Jahren 1974 und 1975 vorgestellten Konzepts der Monitore wurde der erste höherwertige Synchronisierungsmechanismus bereitgestellt, der sich deutlich von Semaphoren [34] und Mutexen unterscheidet. Die herausragende Eigenschaft der Monitore ist in der Bedingung begründet, dass zu jedem beliebigen Zeitpunkt maximal ein *Thread* die im Monitor enthaltene Funktionalität ausführen darf. Versucht ein zweiter *Thread* in einen Monitor einzutreten, in dem sich bereits ein anderer *Thread* in Ausführung befindet, so wird dieser schlafen gelegt und erst dann wieder aktiviert, wenn der erste *Thread* den Monitor verlassen hat.

Abgrenzung

Die *Protocol*-Instanzen des *Gears4Net*-Programmiermodells erfüllen im Zusammenspiel mit den *Schedulern* die Ausführungsbedingung der Monitore, da zu jedem beliebigen Zeitpunkt nur der ausführende *Thread* des der *Protocol*-Instanz zugewiesenen *Schedulers* innerhalb der Instanz aktiv ist. Die Architektur des *Gears4Net*-Modells wirkt sich zudem auf das Verhalten der *Threads* aus. In der klassischen Monitorarchitektur wird ein *Thread* schlafen gelegt, wenn dieser in einen Monitor eintreten möchte, dessen Funktionen bereits von einem anderen *Thread* ausgeführt werden. Dieses Verhalten ist aufgrund der *Gears4Net*-Architektur obsolet. Die ausschließlich Nachrichten-basierte Kommunikation sowie die Zuweisung von genau einem dedizierten *Scheduler-Thread* zu einer *Protocol*-Instanz erwirkt, dass zu keinem Zeitpunkt ein *Thread* in den schlafenden Zustand versetzt werden muss, es sei denn, dass keines der *Protocol*-Instanzen eines *Schedulers* rechenbereit ist.

3.11 Zusammenfassung

Die Entwicklung von massiv parallelen und verteilten Anwendungen, die auf modernen *Multi-Core*-Prozessorarchitekturen ausgeführt werden, stellen die aktuellen Softwareentwicklungsmethoden und Mechanismen vor zwei neue Herausforderungen. Erstens betrachtet keines der praxisrelevanten Programmiermodelle Parallelität als Kriterium des Softwaredesignprozesses, und zweitens greifen all diese Modelle auf die aus den Anfängen der Betriebssystementwicklung bekannten Synchronisierungsmechanismen zurück, deren komplexe Verwendung häufig zu unerwünschten Effekten der Nebenläufigkeit, der Performanzeinbuße oder zu *Dead-Locks* führt.

Der Trend zu ressourcenintensiven Anwendungen wird auch zukünftig durch die Weiterverbreitung von *Multi-Core*-Systemen gestützt. Es bedarf daher der Bereitstellung von geeigneten Programmier- und Architekturdesignmodellen, um einen effektiven und effizienten Softwareentwicklungsprozess zu gewährleisten. Das in dieser Arbeit vorgestellte Programmiermodell *Gears4Net* wurde zu diesem Zweck mit der Zielstellung entwickelt, Parallelität als Designkriterium zu verstehen, auf Synchronisierungsmechanismen zu verzichten und dem Entwickler eine nahezu optimale und Zustandsmaschinen-ähnliche Quellcodestrukturierung zu ermöglichen.

Das *Gears4Net*-Modell führt dazu das Konzept der unabhängigen und parallel auszuführenden *Protocol*-Instanzen ein, die abhängig von den zugrundeliegenden *Scheduling*-Verfahren und den Kommunikationsmechanismen der Instanzen über Prozessbeziehungsweise Rechnergrenzen hinweg verteilt werden können. Die ausschließlich Nachrichten-basierte Kommunikation zwischen den *Protocol*-Instanzen und die Zuweisung eines dedizierten *Schedulers* zu einer *Protocol*-Instanz stellen zudem das Auskommen ohne Synchronisierungsmechanismen sicher.

Das betriebssystemunabhängige Programmiermodell fördert die Strukturierung der Applikationslogik, indem beliebig verschachtelbare Zustandsmaschinen bereitgestellt werden, die beispielsweise die partielle oder vollständige Transformation einer in Automatenschreibweise dargestellten Anwendungslogik in verschachtelte Zustandsmaschinen und Subautomaten ermöglichen. Eine Zustandsmaschine definiert ihren Zustand über die Spezifikation einer *Receiver*-Wartebedingung. Die Spezifikation von Wartebedingungen übernimmt zwei wesentliche Aufgaben innerhalb der *Gears4Net*-Architektur. Erstens ermöglicht sie den Ausdruck komplexer Zustände, beispielsweise des Eingangs von zwei Nachrichten der Typen a und b beziehungsweise einer Nachricht des Typs c oder des Eintritts eines *Timeouts* mit einem vorgegebenen Zeitintervall. Zweitens sind *Receiver*-Wartebedingungen der Architekturbaustein,

der es ermöglicht, trotz des asynchronen und Nachrichten-basierten Verhaltens des *Gears4Net*-Programmiermodells Quellcode zu schreiben, der der Darstellungsform von Zustandsmaschinen-ähnlichem Quellcode aus der blockierenden Programmierung entspricht.

Zusammenfassend lässt sich anmerken, dass das *Gears4Net*-Programmiermodell einen Lösungsweg für den Softwareentwicklungsprozess für massiv parallele und verteilte Anwendungen auf *Multi-Core*-Systemen bietet. Das *Gears4Net*-Programmiermodell liefert zudem die notwendigen Strukturierungsmaßnahmen auf architektureller und programmiertechnischer Ebene und ermöglicht somit einen stringenten Softwareentwicklungsprozess. Die Architektur des *Gears4Net*-Modells ist neben den Ausführungs- und Strukturierungseigenschaften auch für den effizienten Umgang mit eingesetzten Ressourcen verantwortlich. Die ausschließliche Nutzung des asynchronen Programmiermodells mit nicht-blockierenden Methoden und Funktionen ermöglicht dem Modell die maximale Performanz und optimale Skalierbarkeit.

Kapitel 4

Symstry

Gegenstand dieses Kapitels ist das ortsbezogene P2P-System *Symstry* [109]. *Symstry*, dessen Name sich aus der Kombination von *SYMbolic* und *pasTRY* [104] zusammensetzt, unterstützt ortsbezogene Anwendungen mit symbolischen Koordinaten. Häufig sind symbolische Koordinaten für viele Anwendungsfälle besser geeignet als geometrische, da Menschen beispielsweise leichter mit postalischen Adressen als mit Längen- und Breitengraden umgehen können.

Das folgende Kapitel beschreibt nach eingehender Analyse den Aufbau des Systems und die Operationen, die auf dem *Symstry*-Ring ausgeführt werden können. Im Anschluss an die algorithmische Betrachtung wird die Implementierung des Systems auf Basis von *Gears4Net* auszugsweise dargestellt. Den Abschluss findet das Kapitel mit der Beschreibung möglicher Anwendungsszenarien, der Evaluation des Gesamtsystems sowie der Betrachtung der verwandten Arbeiten.

4.1 Motivation

Der zunehmende Erfolg von Navigationsgeräten, Kartendiensten und virtuellen Globen untermauert die immer wichtiger werdende Rolle von ortsbezogenen Anwendungen und Diensten. Die meisten dieser Systeme setzen jedoch entweder auf den Bestand von lokalen Datensätzen, wie dies beispielsweise bei Navigations- oder Geoinformationssystemen der Fall ist, oder stützen sich auf große, kostenintensive Serverinfrastrukturen, wie beispielsweise die Dienste *Google Maps* [41] oder *Microsoft Bing Maps* [73]. Einen gegenläufigen Trend bilden die in letzter Zeit entwickelten P2P-Systeme für ortsbezogene Anwendungen, deren Herausforderung in der Abbildung der zwei- beziehungsweise dreidimensionalen Daten auf ein P2P-System mit einem eindimensionalen Adressraum liegt. Zwei mögliche Lösungen werden von Knoll [62]

und Heutelbeck [50] vorgeschlagen, wobei das erste Verfahren raumfüllende Kurven und das zweite eine *Hypercube*-Struktur nutzt, um eine möglichst exakte Abbildung zu erreichen.

Die vorgestellten P2P-Systeme basieren jedoch allesamt auf geometrischen Koordinaten und beschreiben die Welt mit Längen- und Breitengrad sowie mit einer optionalen Höhenangabe. Geometrische Koordinaten sind jedoch für viele Anwendungsszenarien nicht ideal. Eine Anwendung aus dem Bereich des *Pervasive Computing* [127] könnte beispielsweise für eine Präsentation den nächstgelegenen *Beamer* suchen und dabei einen *Beamer* im Nachbarraum gegenüber einem *Beamer* im Besprechungsraum präferieren, da dessen Standort geometrisch näher ist. Ein Ausweg aus diesem Verhalten kann durch die Einführung des *Geocodings* [9, 66] erreicht werden. Dabei berechnet die *Pervasive Computing* Anwendung zuerst die geometrische Ausdehnung des Raumes und stellt anschließend eine räumlich begrenzte Abfrage. Das theoretisch plausible Verfahren des *Geocodings* stößt jedoch schnell an die Grenzen der praktischen Realisierung, da die wenigsten Benutzer von *Pervasive Computing* Anwendungen die GPS-Daten ihres Hauses beziehungsweise ihres Wohnzimmers kennen. Darüber hinaus sind die Messgenauigkeiten moderner GPS in Gebäuden nicht hoch genug, um eine zuverlässige Bestimmung der Raumdaten zu ermöglichen.

Der Ansatz des *Symstry*-Verfahrens basiert auf der Idee, geometrische durch symbolische Koordinaten zu ersetzen. Dieses Verfahren ermöglicht es, Räume und die darin befindlichen Geräte eindeutig zu beschreiben und anzusprechen. Die Ansprache reduziert sich dabei nicht ausschließlich auf ein dediziertes Gerät sondern auch auf alle Endpunkte innerhalb eines definierten Raumes. Somit ermöglicht *Symstry* beispielsweise die Suche nach allen Rechnern innerhalb eines Hauses, einer Etage oder eines Flures.

4.2 System-Modell und Anforderungsanalyse

Jeder Anwender, der am *Symstry*-P2P System teilnehmen will, muss über einen nicht-mobilen Rechner verfügen, der Rechenleistung und Bandbreite bereitstellt. Das von quasi stationären Rechnern aufgebaute P2P-Netzwerk ermöglicht natürlich auch die Verwendung von mobilen Endgeräten. Diese treten jedoch nicht als Teilnehmer beziehungsweise *Peers* des Netzes sondern als Klienten auf, die das P2P-Netz nutzen. Begründet wird diese Entscheidung durch die vergleichsweise teuren Ein- und Austrittsoperationen. Mobile Rechner müssten sich bei jedem größeren Positionswechsel

abmelden und an anderer Stelle im P2P-Netz einklinken, da die Ordnung der Rechner im *Symstry*-P2P-Netz durch deren physikalische Position bestimmt wird.

Ein wesentlicher Aspekt unseres Systemmodells ist das Lokalitätsprinzip: Jeder Rechner ist für ein Gebiet zuständig, in dem er sich auch selbst befindet. Das ist ein wesentlicher Unterschied zu *Distributed Hash Table (DHT)*-basierten Ansätzen [50]. In DHT-Systemen hat jeder Rechner einen numerischen Identifizierer und ist für alle Daten zuständig, deren Hashwert nahe an diesem liegt. Der Rechner kann daher nicht bestimmen, für welche Daten er verantwortlich sein will. In ortsbezogenen Anwendungen ist das Lokalitätsprinzip dagegen wesentlich, denn ein Rechner, der beispielsweise in Raum 42 steht, erzeugt höchstwahrscheinlich auch die meisten für Raum 42 relevanten Daten. Durch das Lokalitätsprinzip werden die Daten mit hoher Wahrscheinlichkeit dort verwaltet, wo sie auch entstehen, und das schont wiederum die Bandbreite und sorgt für aktuellere Datenbestände.

Zudem löst das Lokalitätsprinzip ein Sicherheitsproblem. Der Besitzer eines Rechners und der daran angeschlossenen Sensoren will für gewöhnlich selbst entscheiden, wem diese Sensordaten zugänglich gemacht werden. Da der Rechner, an den die Sensoren angeschlossen sind, im P2P-Netz auch für sein Gebiet (Raum, Etage oder Grundstück) zuständig ist, werden die Sensordaten lokal gespeichert und verwaltet. Somit behält der Benutzer die volle Kontrolle über seine Daten.

Eine weitere wichtige Anforderung an unser System ist, dass es auch nach einer Netzwerksegmentierung noch funktionieren soll. Wenn beispielsweise die Außenanbindung der Universität oder der DSL-Anschluss eines Hauses ausfällt, dann sollte das P2P-Netzfragment innerhalb der Universität immer noch funktionieren. Das heißt, dass alle Daten, die das Universitätsgelände betreffen, weiterhin innerhalb der Universität abfragbar sind, da diese Daten von Rechnern verwaltet werden, die sich auf dem Universitätsgelände befinden und deshalb noch erreichbar sind. Diese Eigenschaft folgt aus dem Lokalitätsprinzip.

Das P2P-Netz muss allerdings so geartet sein, dass es durch die Netzsegmentierung nicht zerstört wird. Dazu fordern wir, dass eine Nachricht, die von Rechner A zu Rechner B geschickt wird, niemals ein Gebiet G verlassen wird, wenn G die Position von A und B abdeckt. Das von uns vorgestellte System erfüllt diese Bedingung. Das löst wiederum ein weiteres Sicherheitsproblem, weil kein Peer außerhalb des Gebietes G sehen kann, welche Peers innerhalb von G miteinander kommunizieren. Dies ist besonders für kommerzielle Anwendungen in Firmen zwingend erforderlich, zumal viele Sensordaten auch juristischen Beschränkungen unterworfen sind.

4.3 Symbolische Adressen

Jeder Knoten eines P2P-Systems verfügt über einen eindeutigen Identifizierer. Dieser besteht im *Symstry*-P2P-System aus einer symbolischen Adresse und nicht aus einem zufällig gewählten Hashwert, wie dies beispielsweise bei den Systemen *CAN* [101], *Chord* [117] oder *Pastry* [104] der Fall ist.

Eine symbolische Adresse beginnt mit dem Präfix der Struktur *prefixName://* und trennt jede Hierarchiestufe mit einem weiteren Schrägstrich ab. Mithilfe dieser einfachen Produktionsregel lassen sich beliebig tiefe Adresshierarchien erzeugen und anhand ihrer lexikographischen Merkmale ordnen. Der Name des Präfixes hängt von der Anwendungsdomäne der symbolischen Adresse ab. Geographische Adressen beginnen beispielsweise mit dem Präfix *geo://*, während unternehmensorganisatorische Adressen das Präfix *org://* voranstellen.

Eine besondere Eigenschaft der symbolischen Adressen ist, dass jede mittels der Produktionsregel erzeugte Adresse unabhängig von ihrer Domäne der lexikographischen Ordnung unterworfen werden kann. Diese Eigenschaft dient später als Ordnungskriterium innerhalb des *Symstry*-Rings. Die in den beiden Abbildungen 4.1 und 4.2 dargestellten geographischen beziehungsweise organisatorischen Adressen lassen sich somit eindeutig ordnen, und es gilt: $g_2 < g_1 < g_4 < g_3$ beziehungsweise $o_1 < o_3 < o_2 < o_4$.

```
g1: geo://de/nrw/duisburg/bismarckstr/90/bc/3/12/rechts/vs-sat
g2: geo://de/nrw/duisburg/bismarckstr/90/bc/3/12/links/vs-app
g3: geo://de/nrw/duisburg/bismarckstr/90/bc/4/07/vs-tw
g4: geo://de/nrw/duisburg/bismarckstr/90/bc/3/12/vs-mobile
```

Abbildung 4.1: Geographische Adressen

```
o1: org://uni-due/ComputerScience/DistributedSystems/stuff/MartinSaternus
o2: org://uni-due/ComputerScience/DistributedSystems/stuff/TorbenWeis
o3: org://uni-due/ComputerScience/DistributedSystems/stuff/MirkoKnoll
o4: org://uni-due/ComputerScience/NetworkedEmbeddedSystems/MarcusHandte
```

Abbildung 4.2: Organisatorische Adressen

Die hierarchische Struktur der symbolischen Adressen erlaubt zudem die Bestimmung des *Greatest Common Prefix (GCP)* zweier beliebiger Adressen. Das Verfahren bestimmt die Anzahl der identischen Hierarchiestufen, wobei das Präfix zwar Bestandteil der Prüfung ist, aber nicht in die Berechnung der Anzahl der identischen Stufen eingeht. Aus den *GCP* Berechnungen für die Adressen g_1 und g_4 beziehungsweise o_2 und o_4 folgt beispielsweise: $GCP(g_1, g_4) = 8$ und $GCP(o_2, o_4) = 2$.

Der *GCP* erlaubt zudem die Beschreibung der Relation zweier symbolischer Adressen pro Hierarchieebene. Zwei Adressen a_1 und a_2 werden bis zur Hierarchieebene h als *local* bezeichnet, solange gilt: $GCP(a_1, a_2) >= h$. Adressen, die über ein nicht identisches Präfix bis zu definierten Ebene verfügen, werden im Gegensatz dazu als *global* bezeichnet.

Die Wahl der Hierarchiestruktur der symbolischen Adressen obliegt dem Nutzer des Systems. Es ist jedoch anzumerken, dass alle Adressen eines P2P-Systems aus Zwecken der Ordnung einer teilweise vorgegebenen Struktur folgen sollten. Dieses kann beispielsweise durch die Vorgabe einer Ontologie oder klar definierte Regeln erfolgen. Im Falle der Domäne der geographischen Adressen könnte dies bedeuten, dass jede symbolische Adresse mit *Land/Bundesland/Stadt/Straße/Hausnummer* beginnen muss. Der Aufbau einer weiteren Struktur innerhalb des spezifizierten Hauses könnte dann durch den Nutzer des jeweiligen Systems erfolgen.

4.4 Operationen des Symstry-Protokolls

Das *Symstry*-Protokoll unterstützt fünf Operationen, die im Laufe dieses Abschnitts betrachtet werden. Dies setzen sich aus dem *Routing*, dem Ein- und Austrittsprotokoll, der Selbststabilisierung sowie der Fähigkeit zum Versand von *Geo-Cast* beziehungsweise *Range-Queries* zusammen.

4.4.1 Routing

Die Verwendung der bekannten Präfixroutingverfahren [62, 104] setzt Adressen mit konstanter Länge voraus. Da diese Eigenschaft bei symbolischen Adressen mit beliebiger Länge im Gegensatz zu Hash-basierten Verfahren nicht erfüllt werden kann, muss *Symstry* ein eigenständiges *Routing*-Verfahren implementieren. Dieses muss jedoch analog zu den bekannten Präfixroutingverfahren der Eigenschaft genügen, Nachrichten in *O(log n)* Schritten zuzustellen.

Das in *Symstry* verwendete *Routing*-Verfahren basiert auf der aus *Pastry* bekannten Nachbarschaftsliste, dem *Leaf Set*, sowie einem *Routing*-Baum, der Referenzen auf weiter entfernte Knoten hält. Die verwendete Baumstruktur resultiert aus der Forderung der Lokalitätseigenschaften des *Symstry*-Netzwerkes. Diese ermöglicht, die Anzahl der *Routing*-Einträge pro Baumknoten zu beschränken und somit mehr lokale als globale Einträge zu speichern. Der *Routing*-Baum, dessen Knoten als *Routing*

Tree Entries (RTE) bezeichnet werden, unterstützt dabei ausschließlich die Operationen Suchen, Einfügen und Löschen. Der *RTE* selbst ist ein Tupel bestehend aus *Key*, *Reference*, *Timestamp*, *Endpoint* <*k, r, t, e*>, wobei

- k ein Schüssel ist, der angibt, für welches Präfix der *RTE* verantwortlich ist.
- r eine Referenz auf den Knoten mit dem Präfix k ist.
- t der Zeitpunkt ist, zu dem der *RTE* erzeugt beziehungsweise aktualisiert wurde.
- e der Endpunkt bestehend aus *IP*-Adresse und *Port* ist, unter dem der *Routing*-Knoten erreicht werden kann.

Während der Initialisierung eines Knotens p mit Präfix *geo://* wird die Wurzel des *Routing*-Baumes mit dem Wert <*geo://, p, DateTime.Now, null*> belegt. Der Knoten p ist somit bei leerem *Routing*-Baum für das gesamte Präfix *geo://* zuständig. Der Zeitstempel des Wurzeleintrages wird mit der aktuellen Zeit belegt, zu der der *Routing*-Baum erzeugt wurde. Der Endpunkt des Knotens wird mit *null* initialisiert, da der Knoten nicht zu sich selbst *routen* muss. Eine Änderung an diesem initial vergebenen Wert wird nicht mehr vorgenommen.

Der Aufbau eines exemplarischen *Routing*-Baumes findet sich Abbildung 4.3. Die Abbildung stellt die Hierarchietiefe des Baumes durch die Einrückungen der in rot markierten Schlüssel dar. Die im Anschluss an die Schlüssel folgenden Referenzen auf die Konten, die für den Schlüsselraum verantwortlich sind, werden in blau abgebildet. Den Abschluss jeder Zeile bildet das Erzeugungs- beziehungsweise Aktualisierungsdatum, das in grün verzeichnet ist. Auf die Angabe des Endpunktes bestehende aus *IP*-Adresse und *Port* wird aus Darstellungsgründen verzichtet.

Suche des dem Ziel nächstgelegenen Knotens

Um dem Ziel eines möglichst effizienten *Routing*-Verfahrens näher zu kommen, ist es unerlässlich, bei jedem *Routing*-Schritt möglichst nahe an die Zieladresse zu springen. *Symstry* verwendet dazu eine *Routing*-Nachricht, die die Zieladresse sowie die zuzustellenden Daten enthält. Jeder Knoten, der eine *Routing*-Nachricht erhält, durchsucht seinen *Routing*-Baum und sein *Leaf Set* nach dem Knoten, der das Nähekriterium am besten erfüllt. Wurde ein solcher Knoten gefunden, wird die *Routing*-Nachricht an diesen näher liegenden Knoten weitergeleitet. Sollte jedoch kein Knoten gefunden werden, der näher an der Zieladresse liegt als der aktuelle Knoten, ist das *Routing*-Verfahren beendet. Der Knoten sendet eine *Routing-Reply*-Nachricht an den Initiator des *Routing*-Verfahrens.

```
geo:// - geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat - 25.05.10 - 14:12
  geo://de - geo://de/by/muenchen/stadtpark/17/5/wohnzimmer/1 - 25.05.10 - 16:28
    geo://de/bw - geo://de/bw/stuttgart/universitaetsstraße/108/1/2034/links/vs-test1 - 25.05.10 - 16:23
    geo://de/nrw - geo://de/nrw/bochum/gertherstraße/266/kg/srv1 - 25.05.10 - 16:25
      geo://de/nrw/bochum - geo://de/nrw/bochum/gertherstraße/266/eg/station3 - 25.05.10 - 16:33
      geo://de/nrw/duisburg - geo://de/nrw/duisburg/lotharstraße/23 - 25.05.10 - 16:24
        geo://de/nrw/duisburg/bismarkstraße - geo://de/nrw/duisburg/bismarkstraße/90/bc/4/07/vs-tw - 25.05.10 - 16:26
          geo://de/nrw/duisburg/bismarkstraße/90 - geo://de/nrw/duisburg/bismarkstraße/90/bc/ls/12 - 25.05.10 - 16:31
            geo://de/nrw/duisburg/bismarkstraße/90/bc - geo://de/nrw/duisburg/bismarkstraße/90/bc/ls/11 - 25.05.10 - 16:29
              geo://de/nrw/duisburg/bismarkstraße/90/bc/3 - geo://de/nrw/.../bc/3/12/rechts/vs-sat-mobile - 25.05.10 - 16:27
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/01 - geo://de/nrw/.../bc/3/01/pool1 - 25.05.10 - 16:23
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/02 - geo://de/nrw/.../bc/3/02/car/leonie - 25.05.10 - 16:31
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/05 - geo://de/nrw/.../bc/3/05/car/leoni - 25.05.10 - 16:30
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/06 - geo://de/nrw/.../bc/3/06/car/l/Karsten - 25.05.10 - 16:25
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/10 - geo://de/nrw/.../bc/3/10/car/jan - 25.05.10 - 16:24
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/11 - geo://de/nrw/.../bc/3/11/car/kai - 25.05.10 - 16:27
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12 - geo://de/nrw/.../bc/3/12/vs-holzapfel - 25.05.10 - 16:29
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/13 - geo://de/nrw/.../bc/3/13/vs-helling - 25.05.10 - 16:33
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/14 - geo://de/nrw/.../bc/3/14/vs-schuster - 25.05.10 - 16:26
                geo://de/nrw/duisburg/bismarkstraße/90/bc/3/15 - geo://de/nrw/.../bc/3/15/vs-wander - 25.05.10 - 16:26
              geo://de/nrw/duisburg/bismarkstraße/90/bc/4 - geo://de/nrw/.../bc/4/links/krawattenjup - 25.05.10 - 16:27
                geo://de/nrw/duisburg/bismarkstraße/90/bc/4/07 - geo://de/nrw/.../bc/4/07/vs-tw-linux - 25.05.10 - 16:28
                geo://de/nrw/duisburg/bismarkstraße/90/bc/4/08 - geo://de/nrw/.../bc/4/08/p/01 - 25.05.10 - 16:24
                geo://de/nrw/duisburg/bismarkstraße/90/bc/4/09 - geo://de/nrw/.../bc/4/09/p/02 - 25.05.10 - 16:31
                geo://de/nrw/duisburg/bismarkstraße/90/bc/4/10 - geo://de/nrw/.../bc/4/10/p/srv-ateam - 25.05.10 - 16:32
            geo://de/nrw/duisburg/bismarkstraße/91 - geo://de/nrw/duisburg/bismarkstraße/91/ba/mainframe - 25.05.10 - 16:24
            geo://de/nrw/duisburg/bismarkstraße/105 - geo://de/nrw/duisburg/bismarkstraße/105/6/341/srv-v3 - 25.05.10 - 16:28
      geo://de/nrw/duesseldorf - geo://de/nrw/duesseldorf/scheibenstraße/eh - 25.05.10 - 16:27
      geo://de/nrw/essen - geo://de/nrw/essen/hausdykerfeld/52/knollmo - 25.05.10 - 16:24
  geo://gb - geo://gb/london/buckingham palace road/ah - 25.05.10 - 16:29
  geo://us - geo://us/fl/lakewales/abc/hat/1 - 25.05.10 - 16:23
    geo://us/wa - geo://us/wa/redmond/ms/bu/17/alexbr - 25.05.10 - 16:32
      geo://us/wa/seattle - geo://us/wa/seattle/anulb/mob1 - 25.05.10 - 16:27
```

Abbildung 4.3: Symstry Routing-Baum

Das Nähekriterium eines Knotens zu einem anderen muss eindeutig definiert sein, damit das *Symstry*-Netzwerk ein deterministisches *Routing*-Verhalten zeigt. Das folgende Verfahren beschreibt unabhängig von der Struktur des *Routing*-Baumes die Suche nach dem Knoten p_n, der dem Knoten p_d am nächsten liegt.

- Bestimme die Knotenmenge M, deren Knoten p_m die maximale gemeinsame Präfixlänge mit dem Zielknoten p_d haben.
- Wenn $|M| = 1$, wähle den einzigen Knoten der Menge M und terminiere das Verfahren.
- Wenn $|M| > 1$, wähle einen beliebigen Knoten aus M als p_n. Durchlaufe alle weiteren Knoten p_m aus M und bestimme den der Zieladresse nächsten Knoten nach dem folgenden Verfahren:
  - Wenn $p_n < p_m < p_d$, wähle $p_n = p_m$.
  - Wenn $p_d < p_n < p_m$, wähle $p_n = p_n$.
  - Wenn $p_n < p_d < p_m$, wähle $p_n = p_n$.

Das vorgestellte Verfahren betrachtet die Suche nach dem nächstgelegenen Knoten auf einer unstrukturierten Knotenmenge. Die hierarchische Struktur des *Symstry-Routing*-Baumes erlaubt jedoch eine deutlich effizientere Umsetzung der Suchalgorithmen. Der *Routing*-Baum besteht aus *RTE*-Elementen, deren Schlüssel k eindeutig angeben, für welchen Präfix der jeweilige *RTE* und dessen Teilbaum zuständig sind. Die Suche nach dem Knoten p_n, der dem Knotens p_d am nächsten liegt, erfolgt über die Suche nach dem *RTE*-Element rte_k, dessen Schlüssel k den größten gemeinsamen Präfix mit der Adresse des Zielknotens p_d aufweist. Der *Routing*-Baum wird dazu ausgehend von der Wurzel durchlaufen, wobei auf jeder Hierarchieebene jeweils das *RTE*-Element gewählt wird, dessen Schlüssel k den größten gemeinsamen Präfix mit p_d aufweist. Das aufgezeigte Verfahren erzeugt somit einen eindeutigen Pfad von der Wurzel des Baumes bis hin zum Element rte_k. Der Pfad, der die *RTE*-Elemente beinhaltet, die auf der jeweiligen Ebene den größten gemeinsamen Präfix mit der Adresse von p_d besitzen, enthält folglich den Eintrag des Knotens p_n, der p_d am nächsten ist. Im Zuge der Pfaderzeugung kann das *RTE*-Element identifiziert werden, dessen Referenz r den größten gemeinsamen Präfix mit der Adresse des Zielknotens p_d aufweist. Ist die Auswahl aufgrund des *GCP* nicht eindeutig, erfolgt die Selektion anhand des im vorherigen Absatz definierten Verfahrens zur Bestimmung des Nähekriteriums.

Eine detaillierte Beschreibung des Verfahrens ist in Form von Pseudocode in Auflistung 4.1 dargestellt. Das rekursive Verfahren erwartet drei Eingabeparameter, die den aktuell zu verarbeitenden und den der Zieladresse nächstgelegenen *RTE* sowie die Zieladresse übergeben. Das Verfahren wird mit der Angabe des Wurzelknoten rte_{root} für die ersten beiden Parameter sowie der Angabe der Zieladresse initialisiert. Die Terminierung der Rekursion erfolgt, sobald ein *RTE* keine Kinkeinträge besitzt, deren Schlüssel das Zielgebiet genauer beschreiben als der *RTE* selbst.

```
1       /// <param name="cRTE">current routing tree entry</param>
2       /// <param name="nRTE">nearest routing tree entry</param>
3       /// <param name="dest">destination address</param>
4       /// <returns>nearest routing tree entry</returns>
5       procedure RTE GetNearestRTE(cRTE, nRTE, dest)
6       {
7           nGcp = GCP(nRTE.Ref, dest)
8           cGcp = GCP(cRTE.Ref, dest)
9
10          if (cGcp > nGcp) OR ((cGcp == nGcp) AND (nRTE.Ref < cRTE.Ref < dest))
11              nRTE = cRTE
12
13          if (HasChildWithLargerGcp())
14              return GetNearestRTE(GetChildWithLargerGcp(), nRTE, dest)
15
16          return nearestRTE
17      }
```

Auflistung 4.1: Rekursives Verfahren zur Bestimmung des nächstgelegenen Knotens

Das vorgestellte Verfahren wird anhand einer exemplarischen Suche verdeutlicht. Die Grundlage für die Suche bilden der in Abbildung 4.3 dargestellte *Routing*-Baum und die Zieladresse *dest = geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-test*. Vor dem Start der rekursiven Suche, deren einzelne Rekursionsschritte in Abbildung 4.4 dargestellt werden, wird der Algorithmus mit der Wurzel des *Routing*-Baumes initialisiert. Die Abbildung 4.4 zeigt für jeden Rekursionsschritt die Zeile des momentan betrachteten *RTE* sowie dessen Schlüssel und Referenz. Zudem wird pro Rekursionsschritt die Referenz auf den Knoten angezeigt, die der Zieladresse am nächsten liegt. Sobald auf dem Pfad ein Knoten gefunden wird, der näher an der Zieladresse liegt, wird dieses in der Tabelle farblich grün markiert. Die Terminierung erfolg,t nachdem der Durchlauf durch den Pfad mit den Zeilennummern [1, 2, 4, 6, 7, 8, 9, 10, 17] abgeschlossen ist und der *RTE* in Zeile 10 des *Routing*-Baumes mit der Adresse *geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat-mobile* als der Zieladresse nächstgelegener Knoten ausgegeben wurde.

dest:	geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-test
Key:	1: geo://
cRTE:	1: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat
nRTE:	1: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat
Key:	2: geo://de
cRTE:	2: geo://de/by/muenchen/stadtpark/17/5/wohnzimmer/1
nRTE:	1: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat
Key:	4: geo://de/nrw
cRTE:	4: geo://de/nrw/bochum/gertherstraße/266/kg/srv1
nRTE:	1: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat
Key:	6: geo://de/nrw/duisburg
cRTE:	6: geo://de/nrw/duisburg/lotharstraße/23
nRTE:	1: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat
Key:	7: geo://de/nrw/duisburg/bismarkstraße
cRTE:	7: geo://de/nrw/duisburg/bismarkstraße/90/bc/4/07/vs-tw
nRTE:	1: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat
Key:	8: geo://de/nrw/duisburg/bismarkstraße/90
cRTE:	8: geo://de/nrw/duisburg/bismarkstraße/90/bc/ls/12
nRTE:	1: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat
Key:	9: geo://de/nrw/duisburg/bismarkstraße/90/bc
cRTE:	9: geo://de/nrw/duisburg/bismarkstraße/90/bc/ls/11
nRTE:	1: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat
Key:	10: geo://de/nrw/duisburg/bismarkstraße/90/bc/3
cRTE:	10: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat-mobile
nRTE:	10: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat-mobile
Key:	17: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12
cRTE:	17: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/vs-holzapfel
nRTE:	10: geo://de/nrw/duisburg/bismarkstraße/90/bc/3/12/rechts/vs-sat-mobile

Abbildung 4.4: Suche nach dem nächstgelegenen Knoten mithilfe des in Auflistung 4.1 vorgestellten Rekursionsverfahrens

Einfügen eines Knotens in den Routing-Baum

Die zweite auf einem *Routing*-Baum definierte Operation ist das Einfügen von Knoten. Die Operation resultiert entweder im Einfügen eines *RTE* in den *Routing*-Baum, in der Aktualisierung eines bestehenden *RTE* oder in der Verdrängung eines alten *RTE*.

Beim *Routing*-Baum des *Symstry*-Netzwerkes handelt es sich um eine in der Tiefe und Breite beschränkte Datenstruktur. Diese Beschränkungen verhindern erstens das Wuchern des *Routing*-Baumes und ermöglichen zweitens die gezielte Steuerung des Verhältnisses der Aufnahme lokaler und globaler *Routing*-Einträge. Die Tiefenbeschränkung des *Routing*-Baumes steuert die Anzahl der in der Struktur zu speichernden Hierarchiestufen. Die Breitenbeschränkung legt hingegen fest, wie viele Kind-Einträge für einen *RTE* erzeugt werden dürfen. Die maximale Anzahl der Kind-Einträge pro *RTE* ist nicht konstant festgelegt. *Symstry* unterscheidet auf jeder Hierarchieebene des *Routing*-Baumes zwischen Adressen mit den Eigenschaften *local* beziehungsweise *global*. Das Verhältnis zwischen der Adresse des Wurzelknotens

des *Routing*-Baums und der einzufügenden Adresse ermöglicht es, den Lokalitätsanforderungen des *Symstry*-Netzwerkes nachzukommen und auf der jeweiligen Ebene mehr lokale als globale Einträge zu speichern. Dies stärkt das lokale Netzwerkwissen des Knotens.

Das Einfügen eines neuen Knotens p_{new} in die *Routing*-Datenstruktur wird im folgenden Verfahren dargestellt:

- Durchlaufe den Baum auf der Suche nach dem *RTE* r_{parent}, dessen Schlüssel k den maximalen *GCP* mit p_{new} hat.
- Befindet sich bereits ein *RTE* auf dem Pfad von der Wurzel des Baumes zu r_{parent}, der auf den Knoten p_{new} verweist, so wird der Zeitstempel dieses *RTE* aktualisiert und das Verfahren terminiert.
- Andernfalls bestimmt das System, ob es sich bei der einzufügenden Adresse um eine als *local* oder *global* charakterisierte Adresse handelt, und bestimmt anschließend die maximale Anzahl der Kinder, die für den Eintrag r_{parent} zulässig sind.
  - Insofern die Breitenbeschränkung die Aufnahme weiterer *Routing*-Einträge zulässt, wird ein neuer *RTE* erzeugt und an den Knoten r_{parent} angehängt.
  - Für den Fall, dass der *Routing*-Baum nicht mehr über genügend Kapazität verfügt, wird der Knoten auf dem Pfad von der Wurzel zu r_{parent} mit dem ältesten Zeitstempel durch p_{new} ersetzt.

Der vorgestellte Knotenersetzungsalgorithmus, der jeweils den Knoten eines Pfades auswählt, der über den ältesten Zeitstempel verfügt, resultiert aus dem in Kapitel 4.4.4 behandelten Selbststabilisierungsverfahren. Die Knoten des *Routing*-Baumes werden dabei in einem definierten Intervall kontaktiert. Insofern ein Knoten die Anfrage beantwortet, werden die Zeitstempel im *Routing*-Baum angepasst. Der Knoten mit dem ältesten Zeitstempel ist somit mit großer Wahrscheinlichkeit nicht mehr erreichbar.

Löschen eines Knotens aus dem Routing-Baum

Die dritte und letzte Operation, die auf einem *Routing*-Baum definiert ist, ist das Löschen eines Eintrages aus der Datenstruktur. Der *Routing*-Baum des *Symstry*-Netzes unterscheidet implizites und explizites Löschen von Knoten. Die implizite Variante wurde bereits im vorherigen Abschnitt vorgestellt, da ein Knoten aus der

Routing-Struktur verbannt wird, insofern die maximale Kapazität erreicht ist und ein neuer Knoten hinzugefügt werden soll.

Die Operation des expliziten Löschens eines Knotens wird hingegen ausgeführt, wenn ein Knoten das *Symstry*-Netzwerk verlassen möchte und vor dem Austritt sein Umfeld über diesen Schritt informiert. Das Entfernen eines beliebigen Knotens p_{del} aus dem *Routing*-Baum wird im folgenden Verfahren beschrieben:

- Durchlaufe den Baum auf der Suche nach dem *RTE* rte_{del}, dessen Referenzadresse identisch ist mit p_{del}.
- Entnehme rte_{del} inklusive seines Teilbaumes aus der Datenstruktur des *Routing*-Baumes
- Durchlaufe den Teilbaum mittels der Breitensuche [28] und füge alle Kinder von rte_{del} über das Verfahren des Einfügens neuer Knoten der *Routing*-Datenstruktur hinzu.

Das Verfahren des expliziten Löschen nutzt den vorgestellten Algorithmus zum Einfügen neuer Knoten in die Datenstruktur. Es unterscheidet sich jedoch in der Zuweisung des Zeitstempels für die in den *Routing*-Baum eingehängten Knoten. Ein neu erzeugter Knoten wird mit einem aktuellen Zeitstempel versehen. Dies ist beim Wiedereinfügen der Knoten des Teilbaumes nicht der Fall. Diese werden mit ihren bestehenden Zeitstempeln in den Baum eingehängt, da das Entfernen eines Knotens nicht zur automatischen Aktualisierung der Kind-Referenzen führt.

4.4.2 Eintrittsprotokoll

Das Eintrittsprotokoll definiert das Verfahren, mittels dessen ein neuer Knoten einem bestehenden *Symstry*-P2P-Netzwerk beitritt. Voraussetzung für den Beitritt ist die Kenntnis der Adresse mindestens eines Teilnehmers des bereits bestehenden Systems. Hat der beitrittswillige Knoten keinerlei Kenntnis über einen Teilnehmer des Netzes, so kann diese über ein *Bootstrapping*-Verfahren [27, 30, 31, 35, 60, 61] erlangt werden.

Ein neuer Knoten p_n, der über den Knoten p_1 einem *Symstry*-Netzwerk beitreten möchte, folgt dem angegebenen Eintrittsprotokoll.

- Der Knoten p_n sendet eine *Routing*-Nachricht an den Knoten p_1, deren Ziel die Adresse des Knotens p_n ist.

- Die *Routing*-Nachricht wird solange weitergeleitet, bis die Nachricht den Knoten p_d erreicht, der am nächsten an der Adresse von p_n liegt.

- Der Knoten p_d sendet eine *Routing-Reply*-Nachricht an den Knoten p_n und teilt diesem seine Position innerhalb des *Symstry*-Netzwerkes mit.

- Die *Routing-Reply*-Nachricht enthält die während des *Routing*-Verfahrens besuchten Knoten sowie alle Einträge des *Leaf-Sets* des Knotens p_d. Der neue Knoten p_n informiert mittels des übertragenen *Leaf-Sets* seine direkten Nachbarn über seine Existenz und nutzt zudem die während des *Routing*-Verfahrens gewonnenen Netzinformationen zum Aufbau seines *Routing*-Baumes.

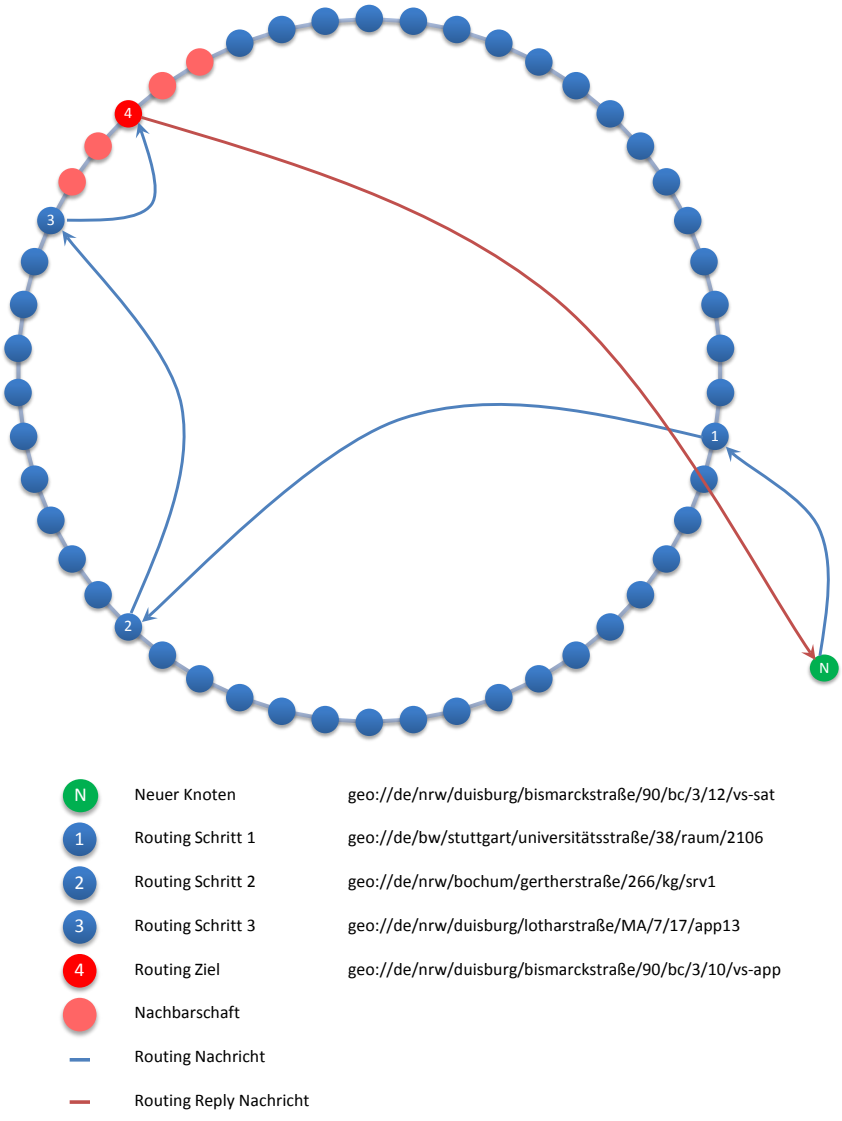

Abbildung 4.5: Eintrittsprotokoll des Symstry-Netzes

Das vorgestellte Eintrittsprotokoll wird am Beispiel des *Symstry*-Netzes in Abbildung 4.5 erläutert. Der grün markierte Knoten p_n mit der Adresse *geo://de/nrw/duisburg/bismarckstraße/90/bc/3/12/vs-sat* beginnt das Eintrittsprotokoll mit dem Versand einer *Routing*-Nachricht an den Knoten p_1 mit der Adresse *geo://de/bw/stuttgart/universitätsstraße/38/raum/2106*. Die *Routing*-Nachricht wird mittels des *Routing*-Verfahrens über die Knoten p_2 und p_3 zum rot markierten Zielknoten p_4 weitergeleitet. Dieser antwortet dem Knoten p_n mittels einer in rot dargestellten *Routing-Reply*-Nachricht und übermittelt darin die Adressen seines *Leaf-Sets*. Zudem beinhaltet die *Routing-Reply*-Nachricht die Adressen der Knoten p_1, p_2, p_3 und p_4, die der Knoten p_n zum Aufbau seines *Routing*-Baumes nutzt.

4.4.3 Austrittsprotokoll

Verlässt ein Knoten das *Symstry*-P2P-System, so kann dieses aktiv oder passiv geschehen. Ein passiver Austritt aus einem Netzwerk findet beispielsweise statt, wenn ein Knoten durch eine Netzwerkpartitionierung vom bestehenden Netz getrennt wird. Dieses Verhalten muss durch die Selbststabilisierungsmechanismen des Netzes aufgefangen werden. Der Prozess des aktiven Austritts folgt hingegen einem definierten Protokoll.

Ein Knoten, der beabsichtigt, das Netz zu verlassen, informiert die im *Leaf-Set* und im *Routing*-Baum enthaltenen Knoten mittels des Versandes einer *Exit*-Nachricht über sein bevorstehendes Ausscheiden aus dem Netzwerk. Erhält ein Knoten eine *Exit*-Nachricht, prüft dieser, ob sich in seinem *Leaf-Set* oder seinem *Routing*-Baum eine Referenz auf diesen Knoten befindet, und entfernt diese. Der Knoten ist nach Abschluss des Verfahrens aus dem Netz ausgeschieden.

4.4.4 Selbststabilisierung

Die Teilnehmer eines P2P-Systems sind naturgemäß einer hohen Fluktuation [31] unterworfen, die entweder durch geringe Onlinezeiten der Teilnehmer oder durch beliebig geartete Netzwerkpartitionierungen hervorgerufen wird. Im Falle eines aktiven Austritts aus dem System informiert der Knoten, der das Netz verlässt, alle ihm bekannten Knoten. Diese Benachrichtigung ist jedoch nicht ausreichend, da die Information „Knoten p_1 kennt Knoten p_2“ nicht automatisch den Umkehrschluss zulässt und somit Knoten, die p_1 nicht kennt, Referenzen auf diesen halten können.

Das passive Verlassen verursacht somit noch mehr tote Referenzen auf nicht mehr existierende Knoten.

Die Stabilisierung des *Symstry*-P2P-Systems setzt daher auf die turnusmäßige Überprüfung der Knoten im *Leaf-Set* und im *Routing*-Baum. Jeder Knoten versendet daher in einem vorgegebenen Intervall eine *Update*-Nachricht an die zu überprüfenden Knoten. Antwortet ein solcher Knoten nicht innerhalb von zwei aufeinanderfolgenden Intervallen, so wird der Knoten als nicht mehr existent markiert und sowohl aus dem *Leaf-Set* als auch aus dem *Routing*-Baum entfernt.

4.4.5 GeoCast

Die fünfte und letzte Operation, die auf einem *Symstry*-Ring definiert ist, ist das Zustellen einer Nachricht an alle Empfänger einer spezifizierten Region. Das als *GeoCast* [37, 58, 102] beziehungsweise allgemeiner als *Range-Query* bezeichnete Verfahren versendet eine *GeoCast*-Nachricht an alle n Knoten im Zielgebiet. Die Anzahl der zuzustellenden Nachricht ist im Netzwerk mit n minimal, da jedem der n Knoten im Zielgebiet eine Nachricht zugestellt werden muss. Der folgende Algorithmus versucht daher, Zustellung der Nachrichten durch die Parallelisierung des Zustellungsverfahrens zu beschleunigen. Ein naiver Algorithmus würde die Nachricht im Zielgebiet von Nachbar zu Nachbar weiterleiten und somit eine Laufzeit von $O(n)$ statt der anzustrebenden $O(log(n))$ aufweisen.

Um das Ziel eines möglichst effizienten Zustellungsalgorithmus zu erreichen, muss dieser das Zielgebiet überschneidungsfrei segmentieren, um eine parallele Zustellung von Nachrichten zu ermöglichen. Der im Folgenden vorgestellte Algorithmus folgt dem *Divide-and-Conquer*-Prinzip und schränkt das effektive Zielgebiet eines Knotens in jedem Schritt weiter ein. Die versandte *GeoCast*-Nachricht enthält deshalb eine durch eine symbolische Adresse repräsentierte untere und obere Schranke, die das Zielgebiet markieren. Eine einfache Intervallhalbierung, die beispielsweise bei *DHT*-basierten Verfahren ihre Anwendung findet, ist aufgrund der Struktur der symbolischen Adressen nicht anwendbar.

Vor dem Versand der ersten *GeoCast*-Nachricht an ein definiertes Zielgebiet ist weder die kleinste noch die größte symbolische Adresse der Zielregion bekannt. Aus diesem Grund kann weder die untere noch die obere Schranke innerhalb der *GeoCast*-Nachricht festgelegt werden. Deshalb prüft jeder Knoten, der eine *GeoCast*-Nachricht mit nicht definierten Schranken erhält, ob er selbst den Rand des Zielgebiets bildet. Der Knoten prüft dazu, ob sein direkter linker beziehungsweise rechter

Nachbar innerhalb des Zielgebietes liegt. Ist dies nicht der Fall, setzt der Knoten sich selbst als obere beziehungsweise untere Schranke. Im nächsten Schritt wird eine lexikographisch geordnete Liste L aufgebaut, die den Knoten selbst sowie alle Knoten des *Routing*-Baumes und des *Leaf-Sets* beinhaltet, die sich innerhalb der Schranken befinden. Mithilfe der Liste werden die Schranken für jeden Knoten berechnet. Der erste Knoten der Liste, L_0, ist beispielsweise für das Intervall zwischen ihm und der unteren Schranke zuständig, der zweite für das Intervall zwischen Knoten L_0 und L_1. Die *GeoCast*-Nachrichten werden im Anschluss an die Berechnungen der Schranken an die jeweiligen Knoten versand. Das Verfahren terminiert, sobald allen Knoten im Zielgebiet eine *GeoCast*-Nachricht zugestellt wurde.

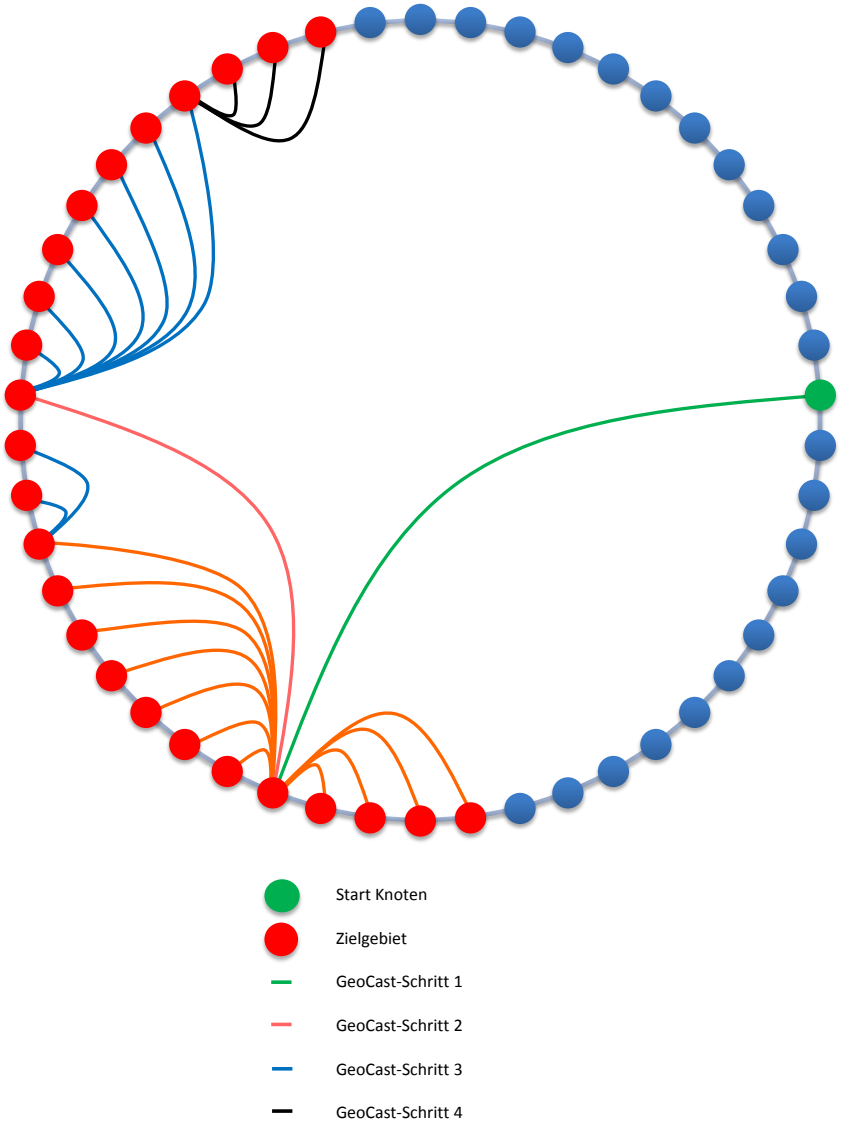

Abbildung 4.6: Versand einer GeoCast-Nachricht

Das beschriebene *GeoCast*-Verfahren erzeugt einen Spannbaum, dessen Tiefe die Laufzeit bestimmt. Eine exemplarische Darstellung des *GeoCast*-Verfahrens wird in Abbildung 4.6 aufgezeigt. Gestartet wird der *GeoCast* durch den grün markierten Startknoten, der als Zielgebiet die rot hinterlegten Bereich spezifiziert. Die erste grün markierte *GeoCast*-Nachricht erreicht einen Knoten inmitten des Zielgebiets. Dieser leitet die eingetroffene Nachricht im zweiten, rot dargestellten Schritt an seine direkten Nachbarn und die Knoten aus seinem *Routing*-Baum weiter. Das Verfahren erreicht bereits im zweiten Zustellungsschritt die untere Grenze des Zielgebiets und dehnt sich in den Zustellungsschritten drei und vier zur oberen Schranke der spezifizierten Region aus. Im vierten und letzten Zustellungsschritt wird die obere Schranke erreicht. Zudem wurden jedem der 25 Knoten des Zielgebietes eine *GeoCast*-Nachricht zugestellt, und das Verfahren terminiert mit einer einer *GeoCast*-Spannbaumtiefe von 4.

4.5 Implementierung des Symstry-Protokolls

Bei der Entwicklung eines P2P-Systems steht neben der Implementierung des *Overlay*-Netzwerkes beziehungsweise der Protokolllogik die Bereitstellung einer geeigneten Infrastruktur im Vordergrund, die die Kommunikation zwischen den einzelnen Konten des Netzwerkes ermöglicht. Die Umsetzung des *Symstry-Overlay*-Netzwerkes fußt auf dem in Kapitel 3 vorgestellten Programmiermodell *Gears4Net* und bindet zudem die ebenfalls auf der Basis des *Gears4Net*-Modells konzipierte Kommunikationsinfrastruktur *Secure Network Abstraction Layer (SNAL)* [123] ein, die im Rahmen des *Peers@Play*-Projektes [121] entwickelt und vorgestellt wurden.

Mit der Entwicklung der *SNAL* wurde eine Bibliothek bereitgestellt, die von der bekannten *Socket*-basierten Kommunikation abstrahiert und eine sichere Kommunikation, wahlweise per *Transmission Control Protocol (TCP)* [97] beziehungsweise *User Datagram Protocol (UDP)* [96], ermöglicht. Die Auswertung der vorherrschenden Netzwerktopologie erlaubt es der *SNAL*-Kommunikationsinfrastruktur zudem, mittels der in [123] beschriebenen Verfahren zwei Endpunkte miteinander zu verbinden, die beispielsweise per *Network Address Translator (NAT)* [116] an ein Netzwerk angebunden sind oder sich hinter einer *Firewall* befinden.

Die Verwendung solcher Verfahren ist bei der Entwicklung von P2P-Systemen unumgänglich, da die Teilnehmer dieser Netzwerke im Regelfall per *NAT* mit dem Internet verbunden sind und zudem meist über eine Firewall verfügen. Diese konzeptionelle Unterscheidung zu klassischen Server-basierten Systemen, bei denen die Gegenstelle

```
1   public partial class SymstryProtocol : ProtocolBase
2   {
3       public override IEnumerator<ReceiverBase> Execute(AbstractStateMachine
            stateMachine)
4       {
5           this.State = OverlayState.Initializing;
6           yield return Launch(InitializeProtocol);
7           yield return Launch(ExecuteProtocol) | exitSignal.CreateReceiver();
8           yield return Launch(CleanUpProtocol);
9           this.State = OverlayState.Disposed;
10      }
11
12      private IEnumerator<ReceiverBase> ExecuteProtocol(AbstractStateMachine
            stateMachine)
13      {
14          this.State = OverlayState.Running;
15          yield return
16              Parallel(Int32.MaxValue, Receive<RoutingMsg>(null,
                    OnRoutingMsgReceived)) |
17              Parallel(Int32.MaxValue, Receive<UpdateMsg>(null,
                    UpdateMsgReceived)) |
18              Interval(this.UpdateLeafSet, 3000, 5000) |
19              Interval(this.UpdateRoutingTree, 3000, 5000);
20      }
21
22      private IEnumerator<ReceiverBase> InitializeProtocol(
            AbstractStateMachine stateMachine)
23      {
24          InitializeSettings();
25          yield return this.initSettingsCompletedSignal.CreateReceiver();
26
27          InitializeP2PLinkManager();
28          yield return this.initLinkManagerCompletedSignal.CreateReceiver();
29
30          InitializeBootstrapper();
31          yield return Receive<BootstrappingInfoMsg>() + HandleBootstrapping;
32      }
33
34      private void UpdateLeafSet()
35      {
36          foreach (RoutingTreeEntry rte in this.leafSet)
37          {
38              this.P2PLinkManager.BeginSend(new LinkMessage(
39                  this.P2PLinkManager.LocalAddress,
40                  rte.LinkAddress,
41                  new UpdateMeg(),
42                  ProtocolType.Symstry), ConnectionBehavior.CreateOnSend);
43          }
44      }
45  }
```

Auflistung 4.2: Auszug aus der Implementierung des Symstry-Protokolls

direkt angesprochen werden kann, muss bei der Entwicklung P2P-basierter Systeme beachtet werden, da beispielsweise der Verbindungsaufbau zweier Endpunkte größeren Zeitverzögerungen unterworfen ist.

Die Symstry-Protocol Klasse

Die Implementierung des *Symstry-Overlay*-Netzwerkes ist auszugsweise in Auflistung 4.2 dargestellt. Die *SymstryProtocol*-Klasse, die wie jede *Gears4Net-Protocol*-Instanz von der Klasse *ProtocolBase* erbt, zeigt die Implementierung des *Overlays* anhand von drei exemplarischen Iteratoren und der Methode *UpdateLeafSet*.

Der Start einer *Symstry*-Instanz bedingt die Ausführung des initialen *Execute*-Iterators (siehe Kapitel 3.6.5), der in den Zeilen 3 bis 10 der Auflistung verzeichnet ist. Dieser begleitet die verschiedenen Zustände, die die *Protocol*-Instanz während ihres Lebenszykluses durchläuft. Jede Instanz startet im Zustand *Created* und überführt diesen in die Zustände *Initializing*, *Bootstrapping*, *Running*, *Leaving* und *Stopped* sowie in den abschließenden Zustand *Disposed*.

Der Eintritt in die Initialisierungsphase der *Protocol*-Instanz wird durch den Aufruf des Teilautomatens *InitializeProtocol* in Zeile 6 der Auflistung 4.2 bedingt. Die dreischrittige Phase beginnt mit der Initialisierung der Konfiguration der *Symstry*-Instanz. Auf den Abschluss des asynchron ausgeführten Konfigurationsprozesses wartet der *InitializeProtocol*-Iterator aufgrund der in Zeile 24 definierten *Signal*-Wartebedingung. Die mittels der Methode *CreateReceiver* aufgebaute Wartebedingung wird genau dann erfüllt, wenn zum Abschluss des Konfigurationsprozesses die Methode *Emit* auf dem *initSettingsCompletedSignal* aufgerufen wird.

Die Schritte zwei und drei der Initialisierungsphase instantiieren und konfigurieren jeweils die Kommunikationsinfrastruktur *SNAL* sowie den *Bootstrapper*, der für das Auffinden eines bereits bestehenden *Symstry*-P2P-Netzwerkes zuständig ist. Die Initialisierung der *SNAL* erfolgt über den Aufruf *InitializeP2PLinkManger* und wartet analog zur Methode *InitializeSettings* mittels eines *Signals* auf den Abschluss des Verfahrens. Die Initialisierung der *Bootstrapper*-Instanz unterscheidet sich lediglich in der Form der Wartebedingung, die nicht auf das Eintreten eines *Signals* sondern auf den Eingang einer *BootstrappingInfoMsg*-Nachricht wartet. Mit dem Eingang der definierten Nachricht wird, wie in Zeile 31 verzeichnet, der Unterautomat *HandleBootstrapping* ausgeführt, der den Quellcode für den Eintritt in das P2P-System enthält und den Abschluss der Initialisierungsphase bildet.

Das *Symstry*-System ist mit Abschluss der Initialisierungsphase im Zustand *Running* angekommen und verbleibt in diesem, bis es von der die *Protocol*-Instanz ausfüh-

renden Anwendung zur Terminierung aufgefordert wird. Die aufgezeigte Bedingung wird durch den *yield*-Ausdruck in Zeile 7 beschrieben. Dieser erzeugt den neuen Teilautomaten *ExecuteProtocol* und wartet auf dessen eigenständige Terminierung oder den Eintritt des *exitSignal*, das von der umhüllenden Anwendung zur Terminierung der *Protocol*-Instanz gesetzt werden kann.

Der *ExecuteProtocol*-Iterator, dessen Quellcode in den Zeilen 12 bis 20 dargestellt ist, erzeugt eine Wartebedingung, die aus vier disjunkten Anweisungen besteht. Die erste Bedingung wartet auf das Eintreffen von Nachrichten des Typs *RoutingMsg* und leitet diese zur weiteren Verarbeitung an die Methode *OnRoutingMsgReceived* weiter. Die zweite Bedingung spezifiziert analog zur ersten Variante eine Wartebedingung, die auf das Eintreffen beliebig vieler *UpdateMsg*-Nachrichten wartet und diese an die Methode *UpdateMsgReceived* weiterleitet. Abgeschlossen wird die sehr kompakte Darstellung der grundlegenden Applikationslogik durch die Einführung zweier *Interval*-Receiver, die beide nach einem initialen Zeitintervall von drei Sekunden in einem wiederkehrenden Turnus von fünf Sekunden ausgeführt werden und die Methoden *UpdateLeafSet* und *UpdateRoutingTree* aufrufen.

Der Quelltext der erstgenannten Methode ist ab Zeile 34 dargestellt und verdeutlicht die Schnittstelle zur Kommunikationsinfrastruktur. Der aufgezeigte Quelltext durchläuft alle Einträge des *LeafSets* und versendet an jeden Eintrag eine *UpdateMsg*-Nachricht über das von der *SNAL* bereitgestellte Objektmodell.

Die auszugsweise dargestellte Implementierung verdeutlicht zum einen die Funktionsweise des im Zuge dieses Kapitels beschriebenen *Symstry*-Protokolls und zeigt zum anderen die Stärken des *Gears4Net*-Programmiermodells, das die sehr einfache und strukturierte Implementierung des Protokolls ermöglicht.

4.6 Anwendungsszenario

Die Anwendungsgebiete für kontextabhängige Applikationen [11, 103], die beispielsweise ortsbezogene Daten verarbeiten, sind vielfältig. Sie werden in diesem Szenario anhand des *Location Based Messengers (LBM)* [111] erläutert, der basierend auf den symbolischen Adressen des *Symstry*-Protokolls Rechner und Regionen adressieren und Einzel- sowie Gruppenchats erzeugen kann. Jeder Teilnehmer des *LBM* verfügt dazu über eine eindeutige symbolische Adresse, die entweder manuell durch den Nutzer eingegeben oder automatisiert von einem Dienst, beispielsweise einem *Lightweight Directory Access Protocol (LDAP)* [114, 124], zugewiesen werden kann.

Der *LBM* stellt basierend auf dem *Symstry*-Protocol drei ortsbezogene Funktionalitäten bereit. Erstens erlaubt er die Adressierung eines Rechners beziehungsweise einer Person anhand ihrer symbolischen Adresse, zweitens ermöglicht die *GeoCast*-Funktion des Systems den Versand einer Nachricht an alle Rechner innerhalb einer Region und den Aufbau eines Gruppenchats, und drittens ermöglicht das System die Einrichtung von Referenzen, wenn sich eine Person an einem anderen Ort als dem initial eingetragenen befindet.

Die Bereitstellung eines *LBM* basierend auf dem *Symstry*-P2P-System kann entweder als vollständig autarke Lösung oder als integraler Bestandteil einer bestehenden *Messenger*-Infrastruktur realisiert werden. Der erste Lösungsansatz bedingt die vollständige Entwicklung des *Messenger-Clients* sowie geeigneter *Bootstrapping*-Verfahren, um dem Netzwerk beizutreten. Der zweite Ansatz realisiert das *Symstry*-System als Adressierungsnetz, das parallel zu der bestehenden *Messenger*-Infrastruktur ausgeführt wird.

Das vorliegende Anwendungsszenario stützt sich auf den zweiten Ansatz und beschreibt die Umsetzung basierend auf den Protokollen des *Microsoft Live Messengers* [86] beziehungsweise des *Jabber-Messengers* [56]. Der Nutzer des *LBM* meldet sich mit seiner *Messenger*-Adresse sowie seiner symbolischen Adresse im System an. Der Eintritt in das P2P-System erfolgt über das Protokoll des *Messengers*. Der *Bootstrapping*-Algorithmus kontaktiert dazu die bereits im Netzwerk integrierten Kontakte aus der *Buddy*-Liste des *Messengers*. Sobald ein *Messenger-Client* dem *Symstry*-P2P System beigetreten ist, kann dieser über das P2P-System ortsbezogene Nachrichten versenden.

Abbildung 4.7 zeigt ein auf dem *Microsoft Live Messenger* basierendes Gruppenchatfenster. Das Chatfenster besteht aus vier Bereichen. Der erste und oberste Bereich gibt das Zielgebiet des Gruppenchats an. Im vorliegenden Beispiel werden alle Teilnehmer, die sich im Gebäude *geo://us/WA/Redmond/R1234/Microsoft Av./19* befinden, adressiert. Der zweite Bereich des Fensters beinhaltet die aktuelle Kommunikation der Teilnehmer, die sich an der angegebenen Lokation befinden. Der dritte und vierte Bereich des Kommunikationsfensters enthält den Eingabebereich für neue Nachrichten sowie eine Karte des Zielgebietes, die zudem die Einbindung von weiteren ortsbezogenen Diensten ermöglicht.

Der Aufbau eines Gruppenchats, wie in Abbildung 4.7 dargestellt, erfolgt über die *GeoCast*-Funktionalität des *Symstry*-P2P-Systems. Der Teilnehmer des Netzwerkes, der eine ortsbezogene Gruppenkommunikation starten möchte, verschickt eine *GeoCast*-Nachricht an das relevante Zielgebiet. Die Nachricht enthält neben der sym-

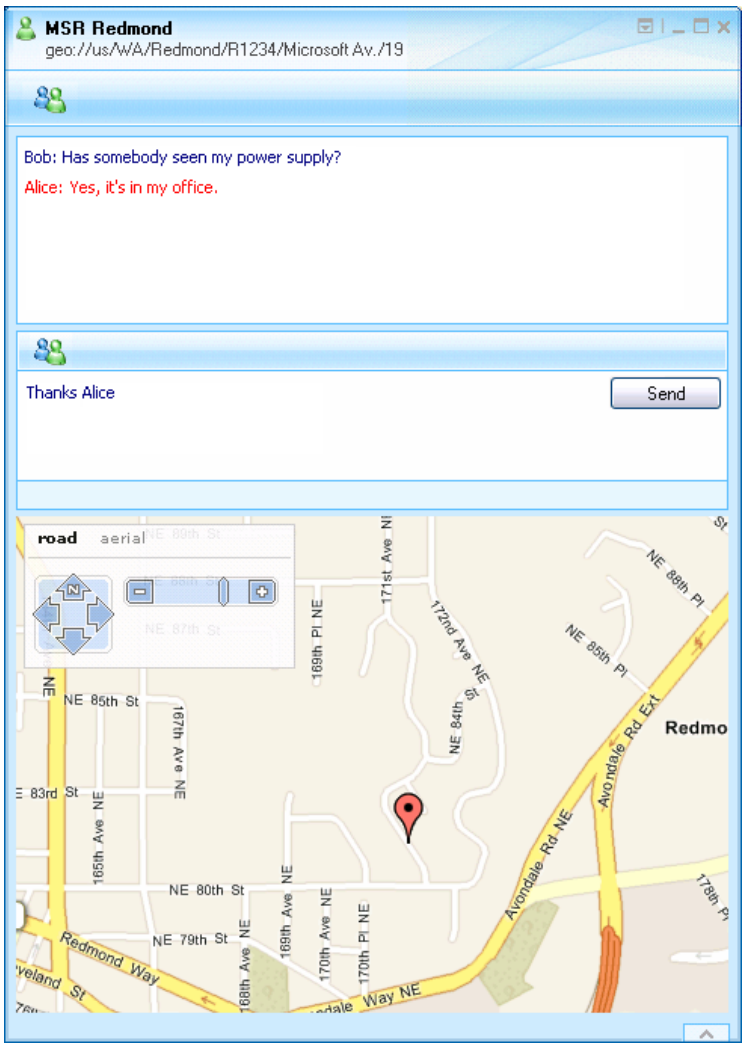

Abbildung 4.7: Lokationsbezogener Messenger

bolischen Adresse des Absenders auch den Identifizierer des verwendeten *Messenger*-Systems. Sobald ein Teilnehmer per *GeoCast*-Nachricht zu einem Gruppenchat eingeladen wurde, kann dieser entscheiden, ob er an diesem teilnehmen möchte. Ist dies der Fall, antwortet der eingeladene Teilnehmer dem Initiator über das *Messenger*-Protokoll und tritt der Kommunikation bei.

4.7 Evaluation

Die Evaluation des *Symstry*-P2P-Systems betrachtet die Effizienz des *Routing*- sowie des *GeoCast*-Verfahrens. Als Bewertungskriterium des *Routing*-Verfahrens wird die Anzahl der durchschnittlichen *Routing*-Schritte des *Overlay*-Netzwerkes auf dem

Weg von der Quelle zum Zielknoten herangezogen. Die Evaluation des *GeoCast*-Verfahrens betrachtet hingegen die Tiefe des erzeugten Spannbaumes als Effizienzkriterium.

Die Messung der beiden Effizienzkriterien erfordert die Bereitstellung eines ausreichend großen *Symstry*-P2P-Netzwerkes. Um dieser Anforderung nachzukommen, ist die Entwicklung einer geeigneten Simulationsumgebung unumgänglich, die zum einen eine große Menge an Knoten erzeugen und zum anderen die bestehende Protokollimplementierung nutzen kann. Die im folgenden beschriebenen Simulationen und Messungen basieren deshalb auf der in Kapitel 5 vorgestellten Simulationsumgebung, die auf Grundlage des *Gears4Net*-Programmiermodells konzipiert wurde und somit die direkte Verwendung der vorgestellten *SymstryProtocol*-Klasse ermöglicht.

Die Simulationsumgebung erzeugt Netzwerke mit einer Knotenanzahl zwischen 500 und 10.000 Knoten. Die symbolischen Adressen werden zufällig generiert und jedem Netzwerkteilnehmer zugewiesen. Der Algorithmus zur Adressengenerierung berücksichtigt jedoch die Messung des *GeoCast*-Verfahrens. Diese kann nur dann ausgewertet werden, wenn das Zielgebiet für einen *GeoCast* bei jeder Messung die identische Größe hat. Aus diesem Grund erzeugt die Adressgenerierung je nach *GeoCast*-Messung 10 beziehungsweise 20 Prozent der Knoten mit einem identischen Präfix auf der ersten Hierarchieebene.

Die Anzahl der Zustellungsschritte sowie die Tiefe des *GeoCast*-Spannbaumes sind abhängig von der Netzkenntnis jedes Knotens. Die Simulationsumgebung wählt für jeden Knoten, der dem Netzwerk beitritt, einen beliebigen, schon im Netzwerk vorhandenen Knoten zum *Bootstrapping*. Sobald alle Knoten in das Netzwerk eingetreten sind, verfügt jeder Knoten über genau die Netzwerkinformationen, die er beim Eintritt in das Netzwerk erhalten hat. Um die Kenntnis des Netzwerkes zu erhöhen und eine natürliche Alterung des Netzwerkes zu simulieren, erzeugt der Simulator *Routing*-Vorgänge, die die *Routing*-Bäume der Teilnehmer füllen. Bei diesem, als *Netzwerkalterungssimulation (NAS)* bezeichneten Verfahren wählt der Simulator für jeden Knoten im Netzwerk einen zufälligen anderen Knoten als Ziel und startet das *Routing*-Verfahren.

Der durchschnittliche Füllstand der *Routing*-Bäume der Knoten eines *Symstry*-Netzwerkes ist in Abhängigkeit von der Netzalterung in Abbildung 4.8 aufgezeichnet. Die Grafik verdeutlicht, dass die Füllstände mit den durchgeführten *NAS*-Verfahren steigen. Die Anzahl der *Routing*-Einträge ist am Ende ausschließlich durch die maximal erlaubte Anzahl der Einträge der *Routing*-Bäume begrenzt.

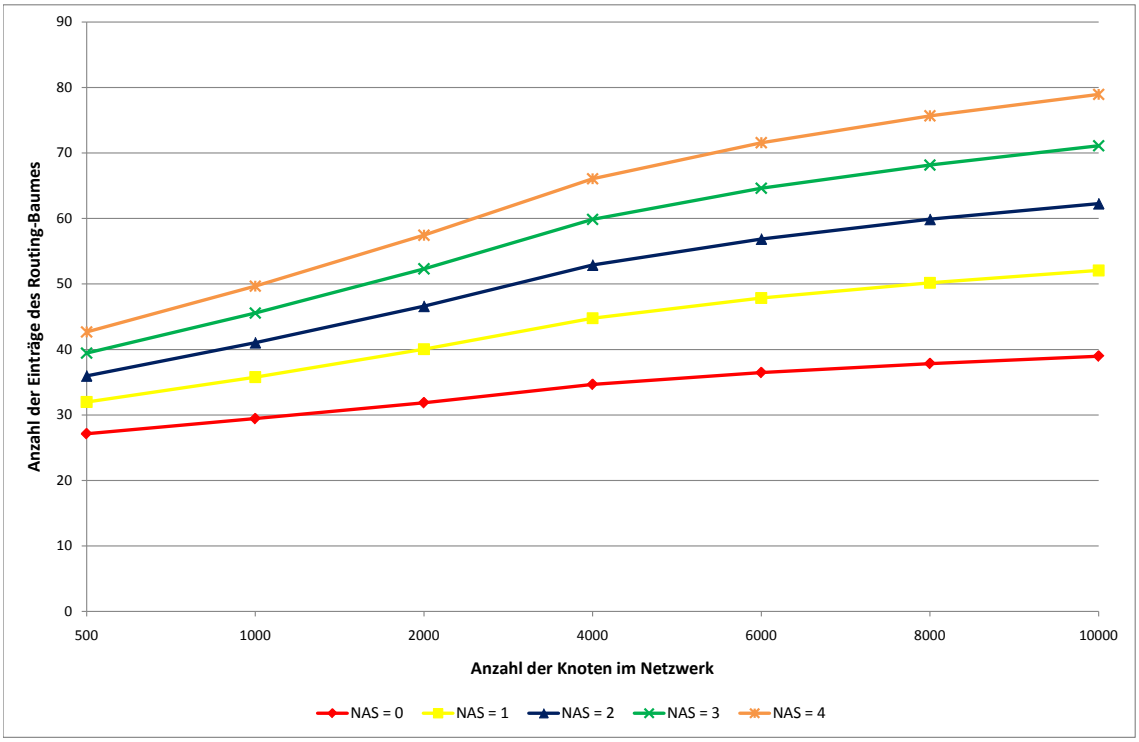

Abbildung 4.8: Anzahl der Einträge des durchschnittlichen Routing-Baumes in Abhängigkeit der Netzwerkalterungssimulation (NAS)

Routingverfahren

Für die Bestimmung der Effizienz des *Routing*-Verfahrens werden Messungen durchgeführt, die die Anzahl der *Routing*-Schritte von einer Quelle zu einem Zielknoten bestimmen. Im Zuge des Verfahrens werden für vier *NAS*-Schritte jeweils 100 Messungen pro Knotenanzahl durchgeführt. Die Anzahl der Knoten pro Netzwerk variiert dabei von 500, 1.000, 2.000, 4.000, 6.000, 8.000 bis hin zu 10.000 Knoten. Die Messungen folgen dabei dem im Anschluss beschriebenen Versuchsaufbau. Jeder Aufbau wird dabei fünfmal durchlaufen.

1. Erzeuge ein *Symstry*-Netzwerk mit n Knoten.
2. Wende das *NAS*-Verfahren je nach Durchlauf 0 bis 4 mal an.
3. Sende eine *Routing*-Nachricht von einem beliebigen Knoten des Netzwerkes zu einem zufällig ausgewählten anderen Knoten und zähle die Anzahl der *Routing*-Schritte, die für die Zustellung der Nachricht nötig gewesen sind.

Die Resultate der Messungen sind in Abbildung 4.9 dargestellt. Das Ergebnis verdeutlicht, dass der Forderung nach einem *Routing*-Verfahren mit $log(n)$ *Routing*-Schritten nachgekommen wird. Das im *Symstry*-P2P-System implementierte *Rou-*

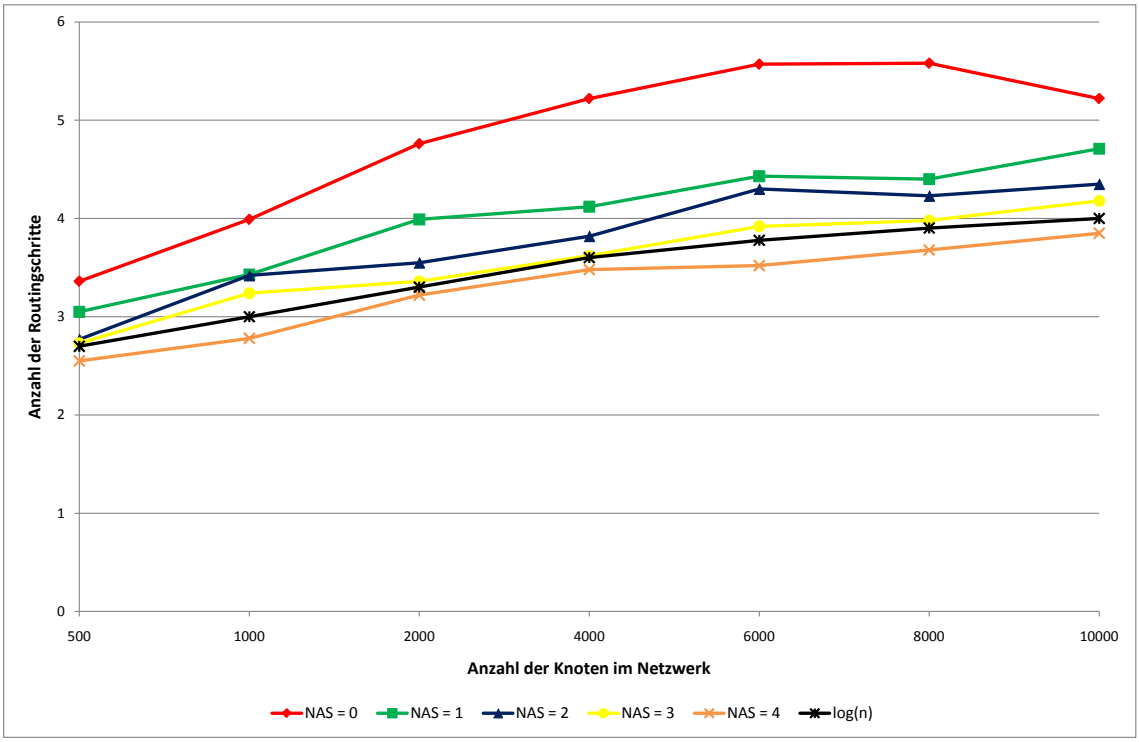

Abbildung 4.9: Anzahl der Routingschritte in Abhängigkeit der Netzwerkalterungssimulation (NAS)

ting-Verfahren erlaubt bereits mit den initialen *Routing*-Informationen ein effizientes Verfahren und nähert sich im dritten *NAS*-Schritt an die in der Farbe Schwarz gekennzeichnete $log(n)$-Kurve an. Ab dem vierten *NAS*-Schritt liegt die Kurve der durchschnittlichen *Routing*-Schritte deutlich unter dieser geforderten Marke und ermöglicht somit ein effizientes *Routing* trotz einer relativ geringen Anzahl an Einträgen im *Routing*-Baum.

GeoCast-Verfahren

Die Bestimmung der durchschnittlichen Tiefe des *GeoCast*-Spannbaumes erfolgt nach einem ähnlichen Messverfahren wie die Bestimmung der Effizienz des *Routing*-Systems. Das Evaluationsverfahren führt wiederum vier *NAS*-Schritte mit jeweils 100 Messungen durch, wobei die Anzahl der Knoten ebenfalls mit dem obigen Messverfahren übereinstimmt. Die durchschnittliche Spannbaumtiefe des *GeoCast*-Verfahren wird durch das fünfmalige Durchlaufen des folgenden Versuchsaufbaus bestimmt.

1. Erzeuge ein *Symstry*-Netzwerk mit n Knoten.
2. Wende das *NAS*-Verfahren je nach Durchlauf 0 bis 4 mal an.

3. Sende eine *GeoCast*-Nachricht von einem beliebigen Knoten an ein beliebiges Zielgebiet, dessen Größe 10% beziehungsweise 20% des gesamten Netzwerkes ausmacht. Es ist darauf zu achten, dass der Initiator des *GeoCast*-Verfahrens nicht innerhalb des Zielgebietes angesiedelt ist.

4. Bestimme die Tiefe des durch den *GeoCast* erzeugten Spannbaumes.

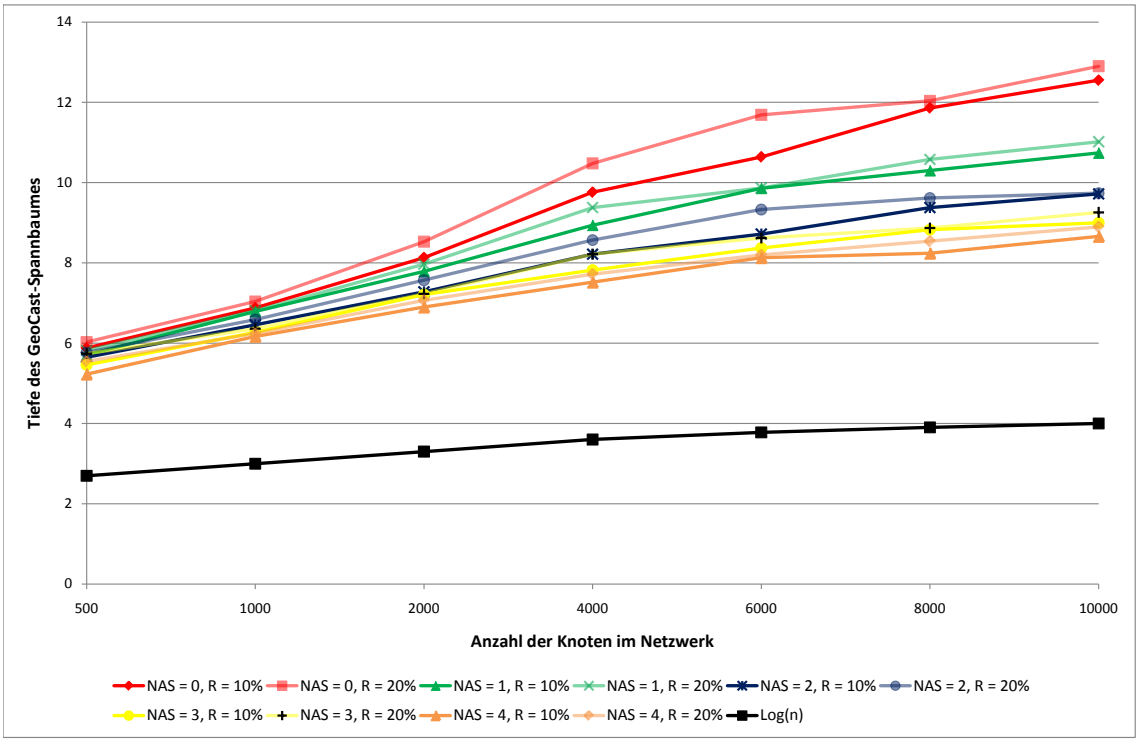

Abbildung 4.10: Tiefe der GeoCast-Spannbaumes in Abhängigkeit der Netzwerkalterungssimulation (NAS) und der prozentualen Größe des Zielgebietes (R)

Die aus der Messung resultierenden, kumulierten Ergebnisse sind in Abbildung 4.10 aufgetragen. Die Messergebnisse bestätigen den bereits beim *Routing*-Verfahren erkennbaren Effizienztrend. Die Messungen für ein Zielgebiet von 10% beziehungsweise 20% nähern sich bis auf einen konstanten Faktor mit zunehmender Netzwerkkenntnis der Idealkurve von $log(n)$ an.

4.8 Weiterentwicklung und weiterer Forschungsbedarf

Bei der Weiterentwicklung des *Symstry*-P2P-Systems stehen zwei Aspekte im Vordergrund. Der erste Aspekt betrachtet das Verhalten des Systems bei einer starken

geographischen Ungleichverteilung der Knoten. Dieses Szenario ist in der Realität sehr wahrscheinlich, da große Städte deutlich dichter besiedelt sind als ländliche Gebiete und somit eine ungleiche Knotenverteilung forciert wird. Es ist daher zu betrachten, welche Auswirkungen die Ungleichverteilung auf das *Routing*-Verfahren des *Symstry*-P2P-System hat. Es ist beispielsweise zu analysieren, ob die Einteilung in lokale und globale Knoten ausreichend ist oder ob eine zusätzliche, feingranulärere Auswertung der Netzwerkstruktur zu verbesserten *Routing*-Ergebnissen führen kann.

Der zweite zu betrachtende Aspekt behandelt das Thema der Netzwerkpartitionierung. Die Lokalitätseigenschaften des *Symstry*-Rings bedingen den Umstand, dass Nachrichten innerhalb eines partitionierten Teilnetzes problemlos zugestellt werden können. Dieses resultiert aus der Kenntnis der direkten Nachbarn. Die Herausforderung, die durch die Lokalitätseigenschaften bedingt wird, besteht in der Zusammenführung eines partitionierten *Symstry*-Rings. Eine mögliche Lösung dieser Problemstellung könnte in der Kombination des *Symstry*-Rings mit einem anderen P2P-System liegen, dessen Optimierungsziel die Gleichverteilung der Identifizierer im Raum ist, und könnte beispielsweise von dem im anschließenden Kapitel vorgestellten *Pastry*-System [104] übernommen werden.

4.9 Verwandte Arbeiten

Das *Symstry*-P2P-System zeichnet sich durch die Verwendung symbolischer Adressen, die Lokalitätseigenschaften sowie die Ringstruktur des Systems und die *GeoCast*-Verfahren aus. Das folgende Kapitel grenzt die bestehenden P2P-Systeme *Pastry* und *Geostry* sowie unterschiedliche Ansätze und Konzepte der *GeoCast*-Verfahren gegenüber *Symstry* ab.

4.9.1 Pastry

Pastry [104] ist ein selbstorganisierendes P2P-System der dritten Generation. Jeder Knoten, der einem *Pastry*-Ring beitreten möchte, berechnet einen eindeutigen *128-Bit*-langen Bezeichner, der die Position des Knotens innerhalb des Rings bestimmt. Die Bestimmung des Bezeichners erfolgt über eine *Hash*-Funktion, die einen eindeutigen Schlüssel basierend auf der *IP*-Adresse des Knotens generiert.

Jeder Knoten k des *Pastry*-Rings verfügt über ein *Leaf Set (L)*, ein *Neighborhood Set (M)* sowie eine *Routing*-Tabelle *(R)*. Das *Leaf Set* beinhaltet eine Liste der numerisch nächsten Knoten, wobei jeweils $\frac{|L|}{2}$ Knoten numerisch kleiner und $\frac{|L|}{2}$ Knoten größer sind als der aktuelle Knoten k. Der Aufbau des *Neighborhood Sets* erfolgt analog zum *Leaf Set*. Es unterscheidet sich jedoch in der Art der gespeicherten Knoten, da im *Neighborhood Set* nicht die numerisch sondern die physikalisch nächsten Knoten abgelegt werden. Die im *Leaf Set* enthaltenen numerisch nahen Knoten sagen somit im Gegensatz zum *Neighborhood Set* nichts über die geographische Nähe der Knoten aus.

Das *Routing*-Verfahren des *Pastry*-P2P-Rings erfolgt über ein klassisches Präfix-*Routing*, bei dem sich eine Nachricht ihrem Ziel in jedem *Routing*-Schritt annähert. Das Zustellungsverfahren einer Nachricht mit dem Ziel d geschieht in folgenden Schritten.

- Durchsuche das *Leaf Set* des Knotens k nach dem Schlüssel d. Existiert ein solcher Schlüssel, dann leite die Nachricht zu Knoten d.
- Ansonsten leite die Nachricht zu dem Knoten aus der *Routing*-Tabelle R, dessen gemeinsames Präfix mit d mindestens um eine Stelle größer ist als das aktuelle gemeinsame Präfix.
- Existiert kein solcher Knoten, leite die Nachricht an einen Knoten weiter, dessen gemeinsames Präfix mindestens genauso lang ist wie das mit dem aktuelle Knoten, der Knoten jedoch numerisch näher am Ziel d ist.
- Existiert kein solcher Knoten, so ist der aktuelle Knoten der Zielknoten. Das *Routing*-Verfahren terminiert.

Die Gemeinsamkeiten zwischen *Pastry* und *Symstry* bestehen vor allem in der Ringstruktur der beiden Systeme. Sie unterscheiden sich jedoch signifikant in ihren Optimierungszielen sowie den verwendeten *Routing*-Mechanismen. Während bei *Pastry* eine möglichst gute Gleichverteilung der Bezeichner im Raum der Identifizierer im Vordergrund steht, optimiert das *Symstry*-P2P-System zugunsten der Lokalitätseigenschaften.

4.9.2 Geostry

Geostry [62] ist ein auf Lokalität optimiertes P2P-System, das aufbauend auf der Idee des *Pastry*-Ansatzes [104] entwickelt wurde. Im Gegensatz zur ursprünglichen

Pastry-Variante beziehungsweise ähnlichen Systemen wie beispielsweise *Scribe* [22] oder *Oceanstore* [63], die die zu speichernden Daten beliebig innerhalb des Netzes ablegen, forciert das Lokalitätsprinzip des *Geostry*-Ansatzes die Datenspeicherung in der direkten geographischen Nachbarschaft. Dieses Verhalten vermindert somit das aufkommende Datenvolumen innerhalb des Netzwerkes.

Geostry erreicht seine Lokalisationseigenschaften durch die gezielte Vergabe der Kontenidentifizierer, die vorgibt, dass zwei Knoten, die sich in geographischer Nähe zueinander befinden, auch das Nähekriterium innerhalb des Raumes der Identifizierer erfüllen müssen. Die gezielte Vergabe der Netzwerkbezeichner erfolgt anhand eines Algorithmus, der basierend auf raumfüllenden Kurven [107] mit besonders optimalen Lokalitätseigenschaften arbeitet.

Die Ausfallsicherheit erreicht das System durch die Replikation der Daten auf mehreren geographisch nahen Knoten. Der dadurch entstehende Datenverkehr ist vergleichsweise gering, da das dem *Overlay*-Netzwerk zugrundeliegende System nur Nachrichten innerhalb der näheren Umgebung zustellen muss.

Die *Routing*-Mechanismen des *Geostry*-Netzes basieren auf einem klassischen Präfixverfahren. Möchte ein Rechner Informationen über einen Ort abrufen, berechnet dieser zuerst den Identifizierer des Ortes, beispielsweise anhand der GPS-Koordinaten, und beginnt anschließend mit dem *Routing*-Verfahren, das die Nachricht an einen geographisch näher gelegenen Knoten weiterleitet. Das Verfahren terminiert, sobald der dem Ziel am geographisch nächsten gelegene aktive Rechner erreicht ist.

Die Gemeinsamkeiten des *Geostry*-Netzwerkes und des *Symstry*-P2P-Systems liegen in der Optimierung der Lokalitätseigenschaften. Die beiden Systeme unterscheiden sich jedoch deutlich in den Adressierungsmethodiken und den *Routing*-Mechanismen. Zudem erschwert das auf raumfüllenden Kurven basierende Verfahren die Berechnung von *Range-Queries*, die in Form von *GeoCast*-Nachrichten sehr effektiv und effizient innerhalb des *Symstry*-Systems definiert und zugestellt werden können.

4.9.3 GeoCast Verfahren

Die Anforderungen zur Realisierung eines *GeoCast*-Dienstes für zehn- oder hunderttausende Benutzer unterscheiden sich signifikant von den in vergleichsweise kleinen geographischen Gebieten operierenden infrastrukturlosen System wie sie in [67] beschrieben werden. Nach Navas und Imielinski [92] können grundsätzlich drei *Geo-*

Cast-Ansätze unterschieden werden: Verzeichnis-basierte Ansätze sowie *Multicast*-basiertes und geographisches Routing.

Verzeichnis-basierte Ansätze stellen Nachrichten durch ein zweistufiges Verfahren zu. Zunächst werden mit Hilfe eines Verzeichnisses, z.B. des *Domain Name Systems (DNS)*, die IP-Adressen der Empfänger der *GeoCast*-Nachricht ermittelt. Im zweiten Schritt werden die Empfänger explizit durch mehrere *Unicast*-Nachrichten adressiert. Obwohl die Empfänger zur Optimierung nach den initialen *Unicast*-Nachrichten einer temporären *Multicast*-Gruppe beitreten können, ist dieser Ansatz nur für kleine Empfängergruppen geeignet.

Beim *Multicast*-basierten Ansatz [55] wird die Welt geographisch partitioniert. Den entstehenden geographischen Gebieten werden *Multicast*-Adressen zugeteilt, denen die Empfänger beitreten. Für eine feingranulare Aufteilung der Welt benötigt dieser Ansatz eine große Zahl an *Multicast*-Gruppen, die erst bei Verfügbarkeit von IPv6 sichergestellt werden kann. Außerdem erweist sich die nur langsam fortschreitende Integration von *Multicast*-Mechanismen in die bestehende IP-Infrastruktur als hinderlich für diesen Ansatz.

Beim geographischen *Routing* treffen die Knoten Weiterleitungsentscheidungen durch den Vergleich des geographischen Zielgebiets und der bekannten geographischen Position der Empfänger. In [92] werden geographische Erweiterungen von gebräuchlichen *Link State* und *Distanz-Vektor-Routingalgorithmen* vorgeschlagen. Im Gegensatz zu diesen Verfahren, die direkt in der IP-Infrastruktur implementiert werden, beschreiben [37, 58, 102] Verfahren, die auf hierarchisch strukturierten *Overlay*-Netzen basieren und damit den wesentlichen Vorteil haben, ohne aufwendige Modifikationen der IP-Infrastruktur auszukommen.

P2P-basierte Ansätze bilden eine interessante Alternative, da sie ohne kostspielige Infrastruktur auskommen. Hierbei wird das *Overlay*-Netz zur Verteilung der Nachrichten verwendet. Eine dedizierte Verteilinfrastruktur eines Anbieters wird nicht mehr benötigt. Idealerweise zeichnet sich ein solches P2P-System durch eine hohe Selbstorganisation und Skalierbarkeit aus. Einen ersten P2P-basierten Ansatz beschreibt Heutelbeck in [51]. Dieser Ansatz beruht auf einer räumlichen Partitionierung und verwendet ein flaches *Routing*-Verfahren, bei dem im wesentlichen Knoten die Nachricht an weitere, im Zielgebiet liegende Knoten weitergeben.

Das *Symstry*-P2P-System und das darin verwendete *GeoCast*-Verfahren grenzt sich jedoch deutlich von dem in [51] vorgestellten Verfahren ab. Während der Ansatz von Heutelbeck die *Distributed data structure (DSPT)* einführt, um geometrische

Objekte effizient verwalten zu können, ermöglicht *Symstry* die Adressierung von *Regionen* über die Angabe einer hierarchisch aufgebauten symbolischen Adresse.

4.10 Zusammenfassung

Das *Symstry*-P2P-System bietet die Plattform für die Entwicklung verteilter ortsbezogener Anwendungen mit symbolischen Koordinaten. Der P2P-basierte Ansatz ermöglicht einen kostengünstigen und skalierenden Einstieg in diese Anwendungsklasse und verzichtet dabei vollständig auf kostenintensive Infrastruktur.

Zudem konnte im Zuge der Evaluation für Netzwerke mit bis zu 10.000 Nutzern gezeigt werden, dass das *Symstry*-P2P-System über ein effizientes *Routing*-Verfahren verfügt und eine Nachricht in $O(log(n))$ Schritten zustellen kann. Die Evaluation hat des weiteren gezeigt, dass das System eine sehr intuitive Definition von *Range-Queries* beziehungsweise *GeoCast*-Regionen ermöglicht und diese mit einem Spannbaum der Tiefe $O(log(n))$ zustellen kann.

Zusammenfassend lässt sich betrachten, dass mit *Symstry* ein für den Bereich der Lokalität optimiertes P2P-System entwickelt wurde, das zudem effiziente *Routing*- und *GeoCast*-Verfahren implementiert. Zudem verdeutlicht der in Auflistung 4.2 dargestellte Ausschnitt des Protokollquellcodes die elegante Implementierung auf Basis des *Gears4Net*-Programmiermodells.

Kapitel 5

Peers@Play-Inspector

Das Kapitel „Peers@Play-Inspector“ bildet den Abschluss des in dieser Arbeit vorgestellten Entwicklungskonzeptes für parallele, verteilte Systeme. Im Anschluss an die Analyse der Architektur von P2P-Systemen und den daraus abgeleiteten Anforderungen beschreibt es das Simulationswerkzeug *Peers@Play-Inspector* inklusive dessen Einsatz- und Konfigurationsmöglichkeiten.

5.1 Motivation

Die zentralen Herausforderungen der Entwicklung paralleler, verteilter Systeme sind in den vorangegangenen Kapiteln detailliert erörtert und anhand des P2P-Systems *Symstry* exemplarisch beschrieben worden. Die Entwicklungsmethodiken wurden dabei durch das in Kapitel *Gears4Net* vorgestellte Programmiermodell *Gears4Net* sowie durch die Kommunikationsinfrastruktur *SNAL* unterstützt, um eine intuitive, synchronisierungsfreie Produktentwicklung zu gewährleisten, die zudem von der unterliegenden Netzinfrastruktur abstrahiert.

Der vollständige Entwicklungszyklus einer solchen P2P-basierten Applikation geht jedoch weit über die Entwicklung des reinen Protokolls hinaus und beinhaltet beispielsweise geeignete Softwaretest- und Analysemechanismen. Im Falle der Entwicklung des *Symstry*-P2P-Systems sind beispielsweise Tests der Netzwerkstabilisierungsalgorithmen sowie Effizienzprüfungen der *Routing*-Verfahren unter möglichst realen Bedingungen erforderlich. Für die Durchführung dieser Tests muss daher ein ausreichend großes Testbett bereitgestellt werden, das tausende P2P-Instanzen ausführen kann und zudem einfache Managementverfahren bietet, die die Analyse des Netzes ermöglichen.

Die Bereitstellung eines solchen Testbetts kann entweder durch eine immens große Anzahl an Rechnern erfolgen, die jeweils eine P2P-Instanz ausführen, oder durch eine Simulationsumgebung, bei der eine große Anzahl an P2P-Knoten auf jedem Rechner, der an der Simulation beteiligt ist, instantiiert wird. Der extreme Managementaufwand sowie die geringe Skalierbarkeit und hohen Kosten des ersten Ansatzes forcieren daher die Betrachtung eines geeigneten Simulators für die Entwicklung und den Test von P2P-basierten Anwendungen.

5.2 Architektur von P2P-basierten Anwendungen

Vor der Entwicklung einer Simulationsumgebung ist die Architektur der zu simulierenden Entität zu betrachten. Ein Architekturmodell, das aus mehreren unabhängigen Schichten besteht, könnte beispielsweise die Simulation jeder einzelnen Schicht ermöglichen und während der Analysephase unterschiedliche Komponenten einer Schicht gegeneinander evaluieren. Ein erster Ansatz für die Schichtenarchitektur wurde von Aberer [1] skizziert. Das in Abbildung 5.1 dargestellte Modell setzt das P2P-*Overlay* direkt auf einer abstrahierenden Netzwerkschicht auf und ermöglicht den direkten Zugriff auf das *Overlay* durch die darüberliegende Schicht des P2P-*Storage* sowie der Applikationsschicht.

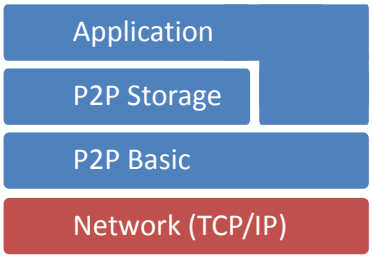

Abbildung 5.1: Schichtenarchitektur einer P2P-Instanz nach Aberer [1]

Ein zweiter, auf dem *ISO/OSI*-Modell aufbauender Ansatz wurde von Weis [126] vorgestellt. Das sehr detaillierte Modell entwickelt ein Schichtenkonzept, das sich ab der fünften Ebene vom bekannten *ISO/OSI*-Modell unterscheidet. Die Schichten, die aufbauend auf dem *Transport-Layer* entwickelt wurden, unterscheiden sich anhand ihrer Adressierungsschema. Jede der fünf in blau gekennzeichneten Ebenen des in Abbildung 5.2 dargestellten Modells verfügt über ein unterschiedliches Adressierungsschema und differenziert sich somit von umgebenden Schichten.

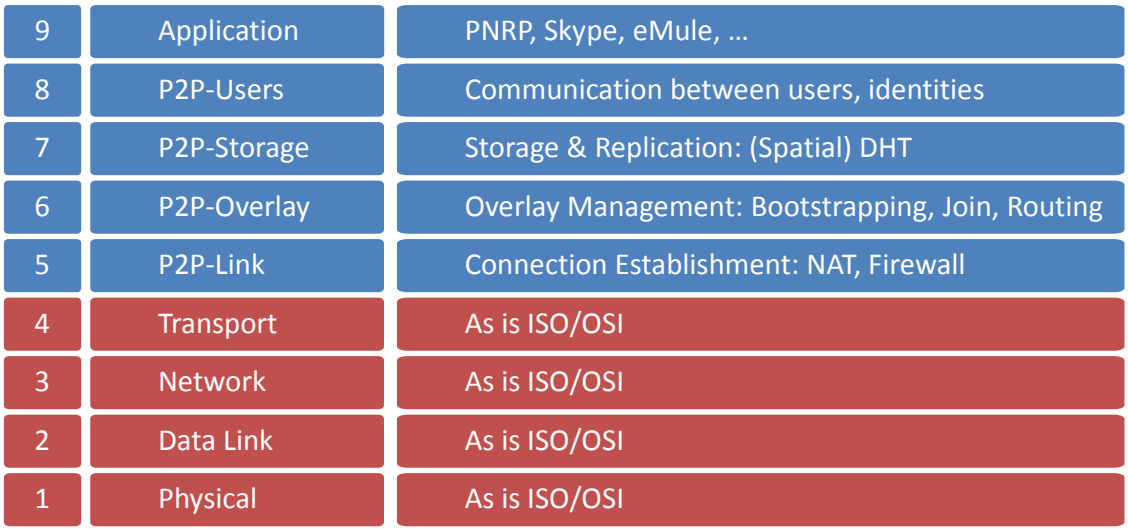

Abbildung 5.2: Schichtenarchitektur einer P2P-Instanz nach Weis [126]

Schicht 5: P2P-Link

Der *P2P-Link-Layer* ist für die Herstellung einer Netzwerkverbindung zwischen zwei P2P-Endpunkten verantwortlich. Die fünfte Schicht des Modells unterscheidet sich dabei vom *Transport-Layer* des *ISO/OSI*-Modells durch die Art des Verbindungsaufbaus sowie die verwendete Adresse. Die Mehrzahl der P2P-Knoten ist nicht direkt sondern über *NAT*-Mechanismen an ein Netzwerk angebunden. Dieser Umstand erfordert die Modifikation des Adressierungsschemas, bei dem die Angabe des Endpunktes (bestehend aus *IP*-Adresse und *Port*-Nummer, wie dies beim *Transport-Layer* der Fall ist) nicht mehr ausreichend ist. Die Adresse wird daher um die interne *IP*-Adresse sowie die zugehörigen internen *Port*-Nummern erweitert und bildet somit ein Tupel aus externer *IP*-Adresse, externer *Port*-Nummer, interner *IP*-Adresse und interner *Port*-Nummer sowie *NAT*- und *Firewall*-Typ.

Schicht 6: P2P-Overlay

Der *P2P-Overlay-Layer* stellt das Protokoll eines jeden P2P-Systems bereit, in dessen Zuständigkeit sowohl *Bootstrapping*-Verfahren als auch der Ein- und Austritt in beziehungsweise aus dem System sowie geeignete *Routing*-Verfahren und Netzstabilisierungsalgorithmen fallen. Die Struktur des Adressierungsschemas, das einen Knoten im Netzwerk eindeutig beschreibt, ist abhängig von der Implementierung des *Overlay*-Netzwerkes. Im Falle der P2P-Systeme *Pastry* [104] und *Chord* [117] werden die Identifizierer beispielsweise durch einen 128-Bit langen *Hash*-Wert, im Falle des auf symbolischen Koordinaten arbeitenden P2P-Systems *Symstry* durch eindeutige symbolische Adressen repräsentiert.

Schicht 7: P2P-Storage

Die siebte Schicht, der *P2P-Storage-Layer*, ist für die Speicherung und das Auffinden von gegebenenfalls redundant verfügbaren Daten innerhalb des P2P-Systems zuständig. Die Adressierung dieser Daten erfolgt dabei über einen Schlüssel, der ein Datum eindeutig definiert. Die Struktur dieses Adressschemas ist wiederum von der Implementierung des *Storage*-Systems abhängig. Im Falle von *DHT*-basierten Verfahren werden die Schlüssel beispielsweise in Form von *Hash*-Werten mit fester Länge angegeben.

Schicht 8: P2P-Users

Die vorletzte Schicht des Modells bildet der *P2P-Users-Layer*, der die direkte Adressierung eines Benutzers über das Adressierungsschema des Benutzernamens ohne den Umweg der Rechneradressierung ermöglicht. Die transparente Schicht ordnet beispielsweise einem Benutzernamen mehrere *Overlay*- und *Storage*-Netzwerke zu und ermöglicht die simultane Anmeldung des Nutzers an mehreren Maschinen.

Schicht 9: Application

Abgeschlossen wird das von Weis vorgestellte Schichtenmodell durch den *Application-Layer*, der die direkte Spezifikation einer Anwendung über einen eindeutigen Applikationsidentifizierer ermöglicht. Anwendungen, die über die neunte Schicht und deren Schemata adressiert werden könnten, sind beispielsweise ePost [87], Skype [10], Scribe [22, 106] oder Past [36, 105] und BitTorrent [98].

5.3 Anforderungsanalyse

Die Anforderungen an eine Simulationsumgebung für P2P-Systeme sind vielschichtig. Das erste Kriterium bildet die maximale Größe des Testbettes. Dies sollte ausschließlich durch den verfügbaren Speicher der ausführenden Maschine und gegebenenfalls durch die vom Betriebssystem vorgegebene Limitierung der Anzahl der Netzwerkverbindungen beschränkt sein. Das Testbett sollte somit im Rahmen der vorgestellten Einflussfaktoren mehrere hundert bis mehrere tausend P2P-Instanzen pro Maschine erzeugen und verwalten können.

Das zweite Anforderungskriterium betrachtet die Architektur der zu simulierenden P2P-Instanzen. Die Simulationsumgebung soll dem Schichtenmodell von Weis [126] folgen und eine P2P-Instanz nicht als einen monolithischen Block sondern als einen Stapel einzelner Schichten verstehen, die beispielsweise zum Zwecke der Analyse

gegeneinander ausgetauscht werden können. Dieses Verfahren ermöglicht unter anderem den Test beziehungsweise die Analyse eines *Storage-Layers* auf verschiedenen *Overlay*-Systemen wie *Pastry* oder *Chord*.

Das dritte Kriterium, das an die Konzeption der Simulationsumgebung gestellt wird, betrachtet die Verwendung des produktiven Quellcodes jeder einzelnen Schicht des Stapels. Diese Forderung verhindert die Divergenz von Produktiv- und Simulationsquelltext und stellt somit sicher, dass der auszuliefernde Produktivquellcode innerhalb der Simulation ausgiebig getestet und analysiert werden konnte.

Das vierte und letzte Anforderungskriterium stellt weitreichende Anforderungen an die Managementfunktionalitäten des Simulators. Die Simulationsumgebung muss ein einfaches Verfahren des *Software-Deployments* unterstützen und zudem einfache Mechanismen zum Hinzufügen und Entfernen von P2P-Knoten zum beziehungsweise aus dem Netzwerk bereitstellen. Die Managementfunktionalitäten müssen sich desweiteren durch die Bereitstellung eines Konfigurationsmechanismus auszeichnen, der für jeden erzeugten Knoten innerhalb der Simulationsumgebung dedizierte Einstellungen bereithält. Abgeschlossen werden die Anforderungen an die Managementfunktionalitäten mit der Forderung nach geeigneten Analysemethoden, die beispielsweise durch speziell aufbereitete *Logging*-Informationen oder *Debugging*-Verfahren unterstützt werden.

5.4 Die Simulationsumgebung Peers@Play-Inspector

Die in der Anforderungsanalyse spezifizierte Simulationsumgebung findet ihre Umsetzung im *Peers@Play-Inspector*, dessen Funktionalität innerhalb dieses Kapitels beschrieben wird. Das Hauptfenster des in Abbildung 5.3 dargestellten Werkzeugs ist in die drei Funktionsbereiche „Netzwerkspezifikation“ am oberen Rand der Abbildung, „Netzwerk-, Analyse- und Instanzmanagement“ am rechten Bildrand sowie die „Nachrichtenverwaltung“ im Zentrum des Hauptfensters untergliedert. Die drei Funktionsbereiche, die im Folgenden vorgestellt und erläutert werden, sind zudem in den Abbildungen 5.4, 5.6 und 5.8 einzeln dargestellt.

Der *Peers@Play-Inspector*, der eine Vielzahl von P2P-Knoten erzeugt, die in verschiedenen *Threads* beziehungsweise von unterschiedlichen *Schedulern* ausgeführt werden, föderiert die Statusänderungen und *Logging*-Informationen der einzelnen Knoten. Jeder schreibende Zugriff auf die Steuerelemente der graphischen Benutzeroberfläche aus den *Threads* oder *Schedulern*, die einen P2P-Knoten ausführen,

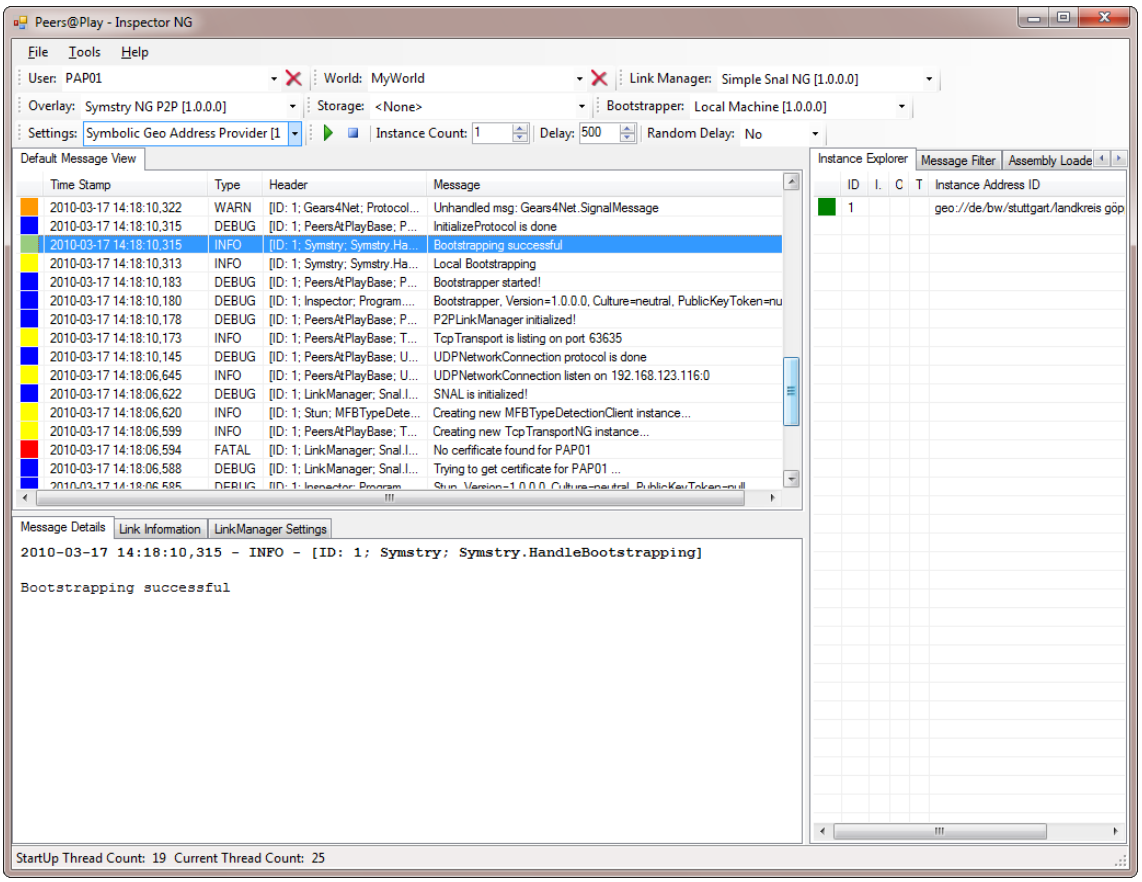

Abbildung 5.3: Peers@Play-Inspector

würde daher eine Synchronisierung mit der Oberfläche nach sich ziehen. Um dieses aufwändige Verfahren zu vermeiden, wurde der *Peers@Play-Inspector* vollständig auf Basis des *Gears4Net*-Programmiermodells entwickelt. Die Anwendung beziehungsweise das Anwendungsprotokoll wird von einem *WindowsFormsScheduler* ausgeführt, der explizit für den Umgang mit graphischen Oberflächen entwickelt wurde. Sobald ein beliebiger Knoten eine Statusänderung anzeigen möchte, stellt dieser eine Nachricht in die Warteschlange des *Inspector-Protocols* ein, die im Kontext des *WindowsFormsScheduler* und somit im *GUI-Thread* verarbeitet wird.

5.4.1 Netzwerkspezifikation

Die Netzwerkspezifikation bildet das Herzstück des *Peers@Play-Inspectors* und erfüllt zugleich die ersten drei Kriterien der Anforderungsanalyse. Die Funktionalität dieses zur detaillierten Erläuterung in Abbildung 5.4 dargestellten Bereiches beinhaltet die Konfiguration und Erzeugung beliebiger P2P-Instanzen.

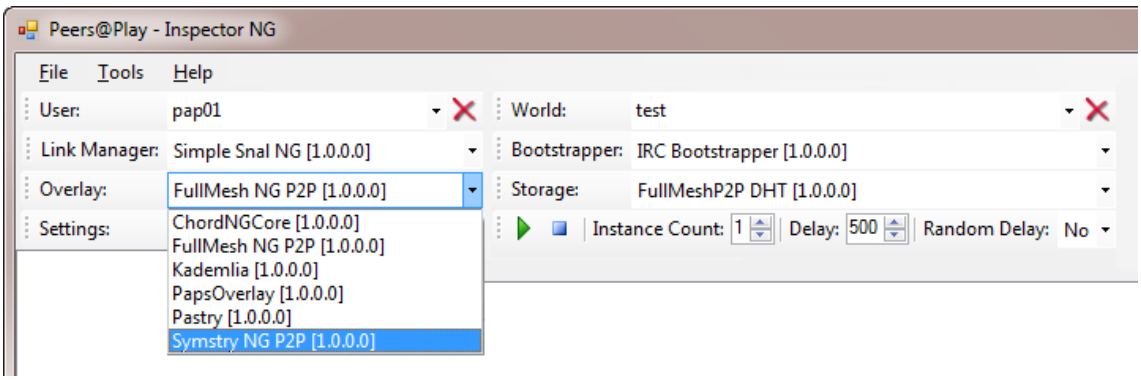

Abbildung 5.4: Netzwerkspezifikationsbereich des Peers@Play-Inspectors

Instanziierung und Terminierung

Der Start der konfigurierten P2P-Instanzen erfolgt über einen Klick auf den grünen *Start*-Kopf, der im unteren Bereich der Abbildung 5.4 dargestellt ist. Die Instantiierungsroutine folgt dabei der gewählten Konfiguration der P2P-Instanzen sowie der Startkonfiguration. Letztere enthält Angaben über die Anzahl der zu erzeugenden Instanzen sowie Informationen über die zeitliche Verzögerung, die zwischen der Erzeugung zweiter Instanzen eingehalten werden soll. Die Verzögerung kann entweder als fester Wert über das Feld *Delay* oder als zufälliger Wert mit einer maximalen Verzögerung in Millisekunden angegeben werden. Die Auswahl einer zufälligen Verzögerung erfolgt über die Auswahlbox *Random Delay*.

Damit der *Peers@Play-Inspector* das erste Anforderungskriterium erfüllen kann, das die Erzeugung eines ausreichend großen Testbettes fordert, setzt dieser auf die Skalierungseigenschaften des *Gears4Net*-Programmiermodells. Der *Inspector* instantiiert vor der Erstellung der ersten P2P-Instanz eine Menge von *STASchedulern*, auf denen die Knoten des Netzwerkes ausgeführt werden. Die Anzahl der bereitgestellten *Scheduler* variiert in Abhängigkeit von der Anzahl der Rechenkerne der ausführenden Maschine. Es werden jeweils 2 *Scheduler* pro *CPU*-Kern erzeugt, auf die die einzelnen *Protocol*-Instanzen der Knoten verteilt werden, um so einen möglichst hohen Gesamtdurchsatz des Systems zu ermöglichen. Aus Gründen der Analysefähigkeit und der *Debugging*-Mechanismen erzeugt der *Peers@Play-Inspector* zudem für jede P2P-Instanz einen eigenen *Application Context* [91], in dem alle Schichten ausgeführt werden. Der *Application Context*, der über einen eindeutigen Identifizierer verfügt, wird als Mantel für alle Schichten einer P2P-Instanz verwendet. Jede Schicht kann somit auch in großen Simulationen eindeutig einer P2P-Instanz zugeordnet werden, ohne dass ein Instanzidentifizierer in jeder Schicht mitgeführt werden muss. Die Ausführung einer P2P-Instanz innerhalb eines solchen *Contextes* hat je-

doch keinerlei Einfluss auf das zugrundeliegende *Scheduling*-Verhalten. Es ist somit weiterhin möglich, die Schichten einer P2P-Instanz auf unterschiedlichen *Schedulern* auszuführen.

Neben der Erstellung der Knoten implementiert das Steuerelement ebenfalls die Terminierungsfunktionalität der erzeugten Knoten, die durch den blauen *Stop*-Knopf realisiert wird. Die Ausführung dieser Operation zieht die Terminierung aller durch die *Inspector*-Instanz erzeugten Knoten nach sich. Die Beendigung einzelner Knoten kann zudem über das Kontextmenü des *Instance Explorers* erreicht werden.

Instanzkonfiguration

Das zweite Anforderungskriterium fordert die Einhaltung des in Abbildung 5.2 dargestellten Schichtenmodells. Eine P2P-Instanz, die innerhalb des *Peers@Play-Inspectors* ausgeführt wird, kann aus verschiedenen Komponenten der einzelnen Schichten zusammengesetzt werden. Die Verifikation, ob eine Komponente einer Schicht zugeordnet werden kann, erfolgt über die Schnittstellen, die im Rahmen des *Peer@Play-Projektes* eindeutig definiert wurden und der Struktur des Schichtenmodells entsprechen. Sobald eine Klasse innerhalb einer Softwarekomponente eine der definierten Schnittstellen implementiert, kann diese einer Schicht des Modells zugeordnet und bei der Komposition einer P2P-Instanz ausgewählt werden.

Der *Peers@Play-Inspector* fordert somit für eine Komponente, die innerhalb der Simulationsumgebung ausgeführt werden soll, die Implementierung einer Schnittstelle und die Entwicklung des Quelltextes auf Basis des *Gears4Net*-Programmiermodells. Insofern beide Bedingungen erfüllt sind, kann eine aus einem Produktivsystem stammende Komponente direkt zur Simulation im *Inspector* herangezogen werden. Das vorgestellte Konzept erfüllt daher das dritte Anforderungskriterium, das mit der Forderung aufwartet, den Original-Quelltext in die Simulation einbringen zu können, um mögliche Divergenzen zwischen Produktivcode und Simulationscode zu verhindern.

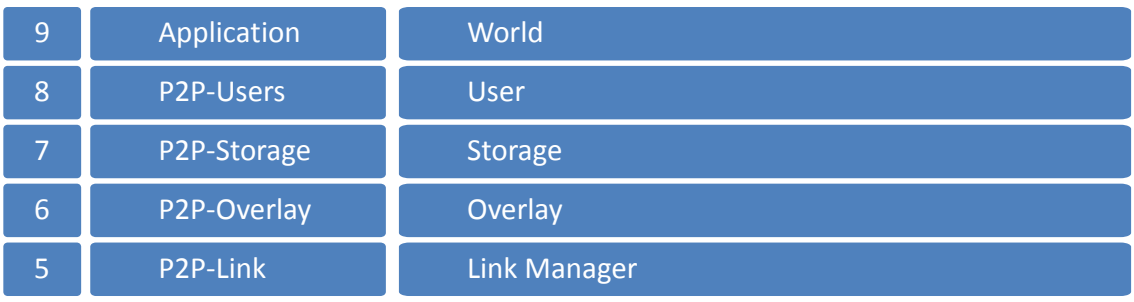

Abbildung 5.5: Abbildung der Inspector-Konfiguration auf das Schichtenmodell

Die Konfiguration einer P2P-Instanz erfolgt über die sieben Schaltflächen aus Abbildung 5.4, von denen fünf dem Schichtenmodell von Weis entsprechen. Die Zuweisung der Schaltflächen zu den einzelnen Schichten des Modells ist in Abbildung 5.5 dargestellt. Die zwei nicht aufgeführten Einstellungen, die Auswahl eines *Bootstrappers* sowie die Zuweisung einer *Settings*-Komponente ermöglichen weiteren Komfort innerhalb der Simulationsumgebung. Die Auswahl des *Bootstrappers*, der nicht im Schichtenmodell enthalten ist und vom *Overlay*-Netzwerk verwendet wird, ist besonders für die Simulation von großer Wichtigkeit, da beispielsweise *Bootstrapping*-Mechanismen unterstützt werden können, die die Simulation auf einem autarken Rechner ohne Netzanbindung ermöglichen. Die Auswahl der *Settings*-Komponente ist für P2P-Systeme relevant, die beim Start mit besonderen Einstellungen versehen werden müssen. Das P2P-System *Symstry* erfordert beispielsweise die Zuweisung einer symbolischen Adresse, die über den *Settings-Provider* vorgenommen wird.

Der *Peers@Play-Inspector* bietet dem Nutzer die Auswahl verschiedener Komponenten pro Schicht, wie dieses am Beispiel der *Overlay*-Auswahlbox in Abbildung 5.4 dargestellt ist. Die Einbindung der unterschiedlichen Komponenten in die Oberfläche des *Inspectors* erfolgt völlig dynamisch und ermöglicht somit ein optimales *Software-Deployment*. Die Aufnahme einer Komponente in eine der Auswahlboxen der Konfigurationsumgebung des *Inspectors* erfordert lediglich die Ablage einer Bibliothek in einem der spezifizierten Verzeichnisse. Der *Peers@Play-Inspector* durchsucht diese Verzeichnisse beim Start der Applikation sowie auf Wunsch des Nutzers und traversiert mittels *Reflection* über die in den gefundenen *Assemblies* enthaltenen Datentypen. Sobald ein Datentyp eine der im *Peers@Play-Projekt* definierten Schnittstellen implementiert, fügt der Inspector diese zur entsprechenden Auswahlbox hinzu.

Um die Lesbarkeit innerhalb der Auswahlboxen zu erhöhen und mögliche Versionsunterschiede offenzulegen, analysiert der *Inspector* die gefundenen Datentypen. Sollten diese über eine *DescriptionAttribute* verfügen, wird die darin enthaltene Beschreibung anstelle des Klassennamens angezeigt. Im Anschluss an den Bezeichner wird zudem die Version der Softwarekomponente angezeigt. Dieses stellt sicher, dass bei der Auswahl unterschiedliche Entwicklungsstände ausgewählt werden können, um diese beispielsweise gegeneinander zu evaluieren.

5.4.2 Netzwerk-, Analyse- und Instanzmanagement

Die Managementfunktionalitäten des *Peers@Play-Inspectors* befinden sich auf der rechten Seite des Hauptfensters in den Registerkarten *Instance Explorer*, *Message Filter* und *Assembly Loader* sowie in den Karten für die Konfiguration der aktuell geladenen *Link Manager*.

Instance Explorer

Der *Instance Explorer*, der vergrößert in Abbildung 5.6 dargestellt ist, listet alle vom *Inspector* instantiierten P2P-Knoten auf und zeigt deren Adresse sowie Informationen über ein- und ausgehende Verbindungen an. Jeder Knoten verfügt zudem über eine farbliche Kennzeichnung, die seinen Status repräsentiert. Ein Knoten wird nach seiner Instanziierung orange markiert und verbleibt in diesem Status, bis das Eintrittsprotokoll vollständig abgeschlossen wurde. Die dritte Zustandsänderung wird beim Austritt des Knotens aus dem P2P-Netzwerk vollzogen. Die Markierung des Knotens folgt dabei dem Zustandswechsel von grün nach rot.

Im Zuge der Simulation können die Knoten des Netzwerkes über den *Instance Explorer* angesprochen und modifiziert werden. Auf einen Knoten kann durch einen Klick auf die entsprechende Zeile im *Explorer* zugegriffen werden. Das Kontextmenü, das in Abbildung 5.6 dargestellt ist, erlaubt es, verschiedene Operationen auf einem Knoten auszuführen. Der erste Menüpunkt des Kontextmenüs mit der Bezeichnung *Log internal state* fordert die ausgewählte *Overlay*- beziehungsweise *Storage*-Instanz auf, eine *Debug*-Ausgabe zu erzeugen, die alle Informationen über den aktuell ausgeführten Knoten enthält. Der Aufruf von *Log internal state* wird durch die Schichtenarchitektur nach unten propagiert und an oberster Stelle föderiert, so dass das in die *Debug*-Ausgabe geschriebene Ergebnis einen Status aller Schichten enthält und somit ein vollständiges Bild über den Zustand des Knotens aufzeigt.

Der zweite Eintrag des Kontextmenüs, die Funktion *Shutdown*, terminiert die ausgewählt P2P-Instanz. Der Knoten wird dabei entsprechend seines Austrittsprotokolls aus dem Netzwerk entnommen und im Anschluss an seine Terminierung farblich markiert.

Der dritte und vierte Eintrag des Kontextmenüs dient der Organisation der Knoten innerhalb des *Instance Explorers*. Sie ermöglichen die Manipulation der Darstellungsreihenfolge der Knoten, die standardmäßig anhand ihres Erzeugungsdatums sortiert

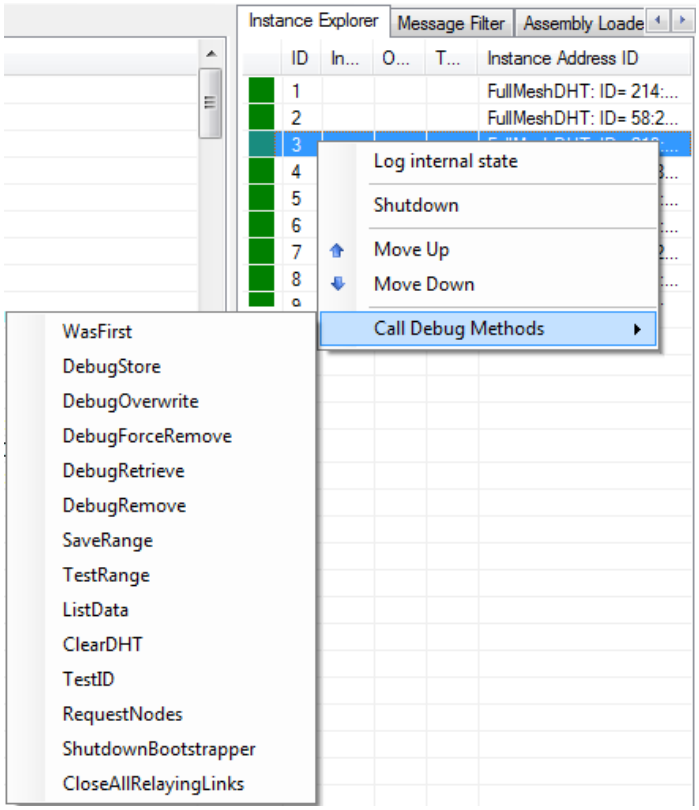

Abbildung 5.6: Darstellung des Instance-Explorers mit zugehörigem Kontextmenü

werden, und schieben die Knoten jeweils eine Stelle nach oben beziehungsweise unten.

Der fünfte und letzte Menüpunkt des Kontextmenüs ermöglicht den Aufruf von Methoden, die zu *Debugging*-Zwecken in die *Overlay*- beziehungsweise *Storage*-Instanz integriert wurden. Bei dem in Abbildung 5.6 dargestellten Unterkontextmenü handelt es sich nicht um eine statische Methodenliste, sondern um ein dynamisch erzeugtes Menü, das während der Auswahl eines Knotens generiert wird. Der *Peers@Play-Inspector* analysiert dazu die Typinformationen der markierten P2P-Instanz. Das *Microsoft .NET Framework* lässt für jeden beliebigen Typ die Abfrage der Typinformationen zu, die unter anderem eine Liste der implementierten Methoden beinhaltet. Der *Inspector* traversiert dazu über die mittels *Reflection* bereitgestellte Liste der Methoden, die für jeden Eintrag ein *MethodInfo*-Objekt bereitstellt. Die Aufnahme einer Methode in das Kontextmenü ist abhängig von zwei Kriterien. Erstens muss die Methode über eine Signatur verfügen, die keine Parameter erwartet und als Rückgabetyp *void* implementiert, und zweitens wird zwingend vorausgesetzt, dass die Methode mit dem Attribut *DebugAttribute* gekennzeichnet ist.

Wählt ein Nutzer eine im Kontextmenü enthaltene Methode aus, so wird diese auf der aktuellen P2P-Instanz aufgerufen. Der durchzuführende Aufruf wird wiederum über den Umweg des *Reflection*-Verfahrens durchgeführt, das auf einem *MethodInfo*-Objekt die Methode *Invoke* aufruft und die P2P-Instanz als Parameter übergibt. Der Aufruf einer *Debug*-Methode resultiert in der Regel in der Ausgabe einer *Debug*- oder *Logging*-Nachricht, die von der Nachrichtenverwaltung dargestellt wird.

Message Filter

Die zweite Reiter im Bereich der Managementfunktionalitäten beeinflusst die Eigenschaften der Nachrichtenverwaltung. Die Registerkarte *Message Filter* ermöglicht es, die Anzahl der anzuzeigenden Nachrichten sowie die Anzahl der zu puffernden Nachrichten festzulegen und zudem benutzerdefinierte Filterkriterien für Nachrichten zu erzeugen.

Die im Nachrichtenfenster des *Peers@Play-Inspectors* dargestellten *Logging*-Nachrichten können einen der sechs Zustände *Debug*, *Info*, *Warn*, *Error*, *Fatal* oder *Other* einnehmen und verfügen zudem neben dem Nachrichten-Inhalt noch über eine Beschreibung ihres Entstehungsortes. Dieser beinhaltet die Information, in welcher Methode einer P2P-Instanz die *Logging*-Nachricht erzeugt wurde. Die Filterkriterien des *Inspectors* erlauben es nun, Nachrichten, die einem definierten Filter entsprechen, im Nachrichtenfenster ein- beziehungsweise auszublenden. Das Filterkriterium kann dabei sowohl einen der sechs Zustände als auch Informationen über den Entstehungsort beinhalten. Es ist beispielsweise möglich, einen Filter zu erzeugen, der keine Nachrichten zulässt, bei denen der Nachrichtenzustand dem Wert *Info* entspricht und die Nachricht von einem P2P-Knoten mit der Instanz-ID 1 stammt.

Die Anwendung der Filterkriterien sind für das Auffinden von Fehlfunktionen innerhalb des Netzes unverzichtbar, da die Darstellung aller *Logging*-Nachrichten aufgrund der Nachrichtenzahl die Konzentration auf konkrete Fehlerszenarien verhindert würde.

Assembly Loader

Die Registerkarte *Assembly Loader* ist für die Konfiguration der Verzeichnisse zuständig, in denen der *Peers@Play-Inspector* nach den *Assemblies* suchen soll, die Komponenten des Schichtenmodells enthalten. Die Abbildung 5.7 enthält zwei Funktionsbereiche. Der obere Bereich ermöglicht die Angabe der Verzeichnisse, die im Rahmen des Suchverfahrens betrachtet werden sollen, während der untere die ge-

fundenen *Assemblies* auflistet und in der tabellarischen Struktur weiteren Aufschluss über deren Eigenschaften gibt.

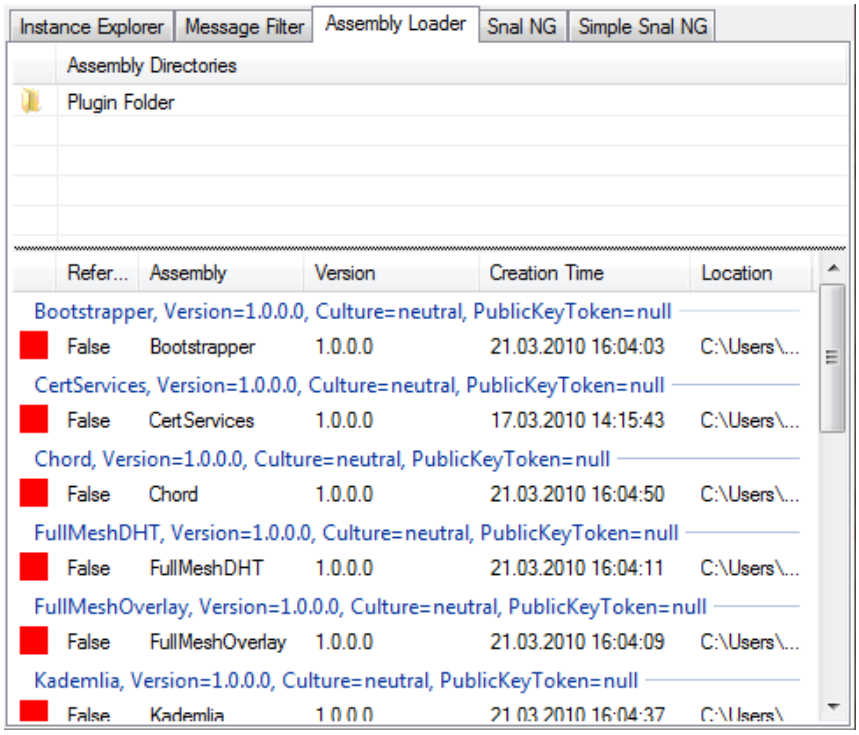

Abbildung 5.7: Registerkarte Assembly-Loader

Die farbige Markierung zu Beginn jeder Zeile kennzeichnet, ob bereits ein Datentyp aus der entsprechenden *Assembly* geladen wurde. Ist dies der Fall, wird eine grüne Markierung vorgenommen, andernfalls eine rote. Die weiteren Eigenschaften, die der Tabelle zu entnehmen sind, beschreiben beispielsweise den vollständigen *Strong-Name* einer *Assembly* und alle darin enthaltenen Datentypen sowie die vorliegende Versionsnummer und den Ort, von dem die *Assembly* geladen wurde.

Die Aufgabe des *Assembly Loaders* ist aus zwei Gründen hervorzuheben. Erstens ermöglicht der *Loader* die Spezifikation der Orte, von denen die Bibliotheken geladen werden sollen und vereinfacht damit den Prozess der *Software*-Verteilung in großen Simulationen. Zweitens informiert die Registerkarte den Benutzer über die tatsächlich herangezogenen Bibliotheken und warnt beispielsweise bei der Verwendung von identischen Datentypen in unterschiedlichen Versionen.

Link Manager

Den Abschluss dieses Abschnittes bilden die Registerkarten der *Link Manager*, die

dynamisch zur Laufzeit erzeugt werden. Die *Link Manager*, die dem *Peers@Play-Projekt* entspringen, verfügen in der Regel über eigene Konfigurationseinstellungen, die beispielsweise die Transportprotokolle beeinflussen und die Auswahl zwischen *TCP* und *UDP* ermöglichen. Sobald ein der Schicht 5 entsprechender *Link Manager* beim Laden der Datentypen erkannt wird, evaluiert der *Peers@Play-Inspector*, ob dieser über eine graphische Konfigurationsoberfläche verfügt, und stellt diese gegebenenfalls als weitere Registerkarte in den Managementbereich ein.

5.4.3 Nachrichtenverwaltung

Den dritten und letzten Funktionsbereich, der innerhalb des Hauptfensters dargestellt wird, bildet die Nachrichtenverarbeitung, die eingehende *Logging*-Nachrichten anzeigt. Die Kriterien, die zur Aufnahme einer Nachricht in das in Abbildung 5.8 dargestellte Nachrichtenfenster führen, sind von den im Reiter *Message View*-Fenster spezifizierten Filtern abhängig. Das *Message View*-Fenster stellt die eingehenden Nachrichten mit einer farblichen Markierung dar, die Aufschluss über den Typ der Nachrichten gibt. Beispielsweise werden Nachrichten des Typs *Debug* in Blau, Warnungen in orange und Fehler in rot dargestellt.

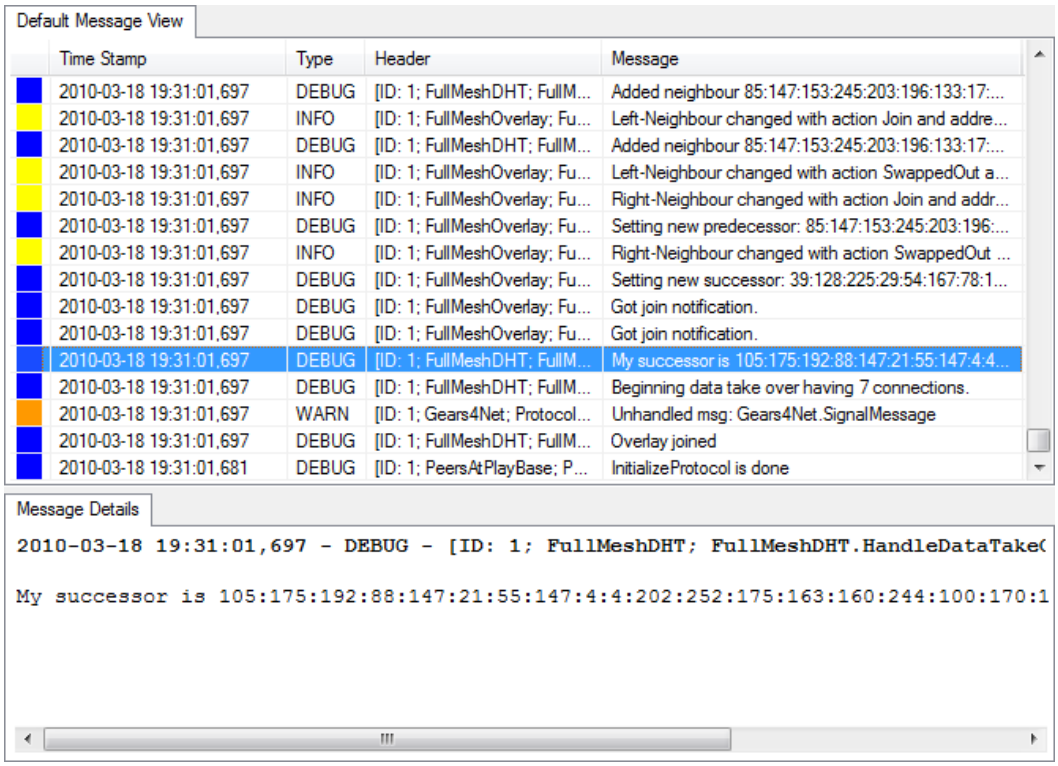

Abbildung 5.8: Darstellung der eingehenden Nachrichten im Message- und Details-View

Die Nachrichtenverwaltung verfügt neben dem *Message View* im oberen Bildbereich noch über den *Details View*. Dieser erlaubt die detaillierte Darstellung der in der aktuell ausgewählten Nachricht enthaltenen Information. Die beiden Nachrichtenfenster ermöglichen somit im Zusammenspiel mit den aufzustellenden Filterkriterien einen optimalen Einblick in die Vorkommnisse der einzelnen Knoten in der Netzwerksimulation.

5.5 Zusammenfassung und Ausblick

Der *Peers@Play-Inspector* ist das Simulationswerkzeug für P2P-Applikationen, die auf Basis des *Gears4Net*-Programmiermodells sowie des von [126] vorgestellten Schichtenmodells konzipiert und entwickelt wurden. Das Werkzeug implementiert die im Zuge der Anforderungsanalyse definierten Kriterien und ist somit in der Lage, große Testsysteme zu erstellen und zu managen. Ein besonderer Fokus wurde während der Entwicklung auf die Analyse von *Logging*-Ausgaben gelegt, die im *Message View* dargestellt und mit verschiedensten Filtern belegt werden können.

Der *Peers@Play-Inspector* ist als offenes, erweiterbares Werkzeug entwickelt worden, das mit den Test- und Analyseanforderungen der Nutzer wachsen kann. Die *Late Binding*-Mechanismen erlauben die schnelle Erweiterung der Anwendung mit neuen Komponenten und Funktionen.

Kapitel 6

Fazit

Der Betrieb und die Wartung skalierender, hochverfügbarer *Client-Server*-Architekturen erfordern eine permanente Investition des Dienstbetreibers und führen somit zu unausweichlichem Kostendruck. Eine Alternative, die diesem Trend entgegentritt, bieten dezentral organisierte *Peer-to-Peer*-Systeme. Diese verlagern die Investition des Dienstbetreibers auf die Nutzer des Dienstes und ermöglichen somit den Betrieb eines kostengünstigen, aber extrem skalierbaren Systems. Der Betrieb von *Peer-to-Peer*-Systemen, der besonders durch die zunehmende Verbreitung von leistungsstarken, parallel arbeitenden Prozessoren und die Zunahme stationärer und mobiler Breitbandnetze forciert wird, hat jedoch erhebliche Auswirkungen auf die Komplexität des Softwareentwicklungsprozesses.

Die vorliegende Arbeit hat daher die Herausforderungen bei der Entwicklung massiv paralleler, verteilter Systeme betrachtet und geeignete Verfahren und Modelle zur Komplexitätsreduktion des Softwareentwicklungsprozesses abgeleitet. Die gewonnenen Erkenntnisse sind in die Entwicklung des *Gears4Net*-Programmiermodells eingeflossen, das die implementierungstechnische Grundlage für das ortsbezogene *Peer-to-Peer*-System *Symstry* bildet. Den Abschluss findet die Arbeit in der Bereitstellung der Simulationsumgebung *Peers@Play-Inspector*, die den Test und die Simulation großer *Peer-to-Peer*-Systeme ermöglicht.

Den ersten Schwerpunkt der drei Themengebiete bildet die Konzeption und Implementierung eines Programmier- und Designmodells für die Entwicklung massiv paralleler, verteilter Systeme. Im Rahmen des Kapitels *Gears4Net* wurden daher bestehende Softwareentwicklungsmethoden im Hinblick auf die multidimensionale Entwicklung bei der Prozessortechnologie in den Dimensionen der zunehmenden Taktraten und dem zunehmenden Parallelisierungsgrad betrachtet und die Relevanz der jeweiligen Methoden auf die Entwicklung der Anwendungsklasse der massiv

parallelen, verteilten Systeme analysiert. Im Anschluss an die Analyse der bestehenden Programmiermodelle und Softwareentwicklungsmethoden, die in verschiedensten Punkten Defizite bei der Implementierung der beschriebenen Anwendungsklasse aufwiesen, erfolgte die Konzeption des *Gears4Net*-Programmiermodells. Dieses kombiniert die Eigenschaften des Aktorenmodells mit der Effizenz des asynchronen Programmiermodells. Die Implementierung des *Gears4Net*-Programmiermodells erfolgte in der standardisierten Programmiersprache *C#* und öffnet sich damit einer besonders großen Gruppe an Softwareentwicklern. Das *Gears4Net*-Modell, das die Betrachtung von Parallelität als Designkriterium ermöglicht, forciert zudem die synchronisationsfreie Anwendungsimplementierung aus der Sicht der Softwareentwickler. Den Abschluss findet die Betrachtung des ersten Themenschwerpunktes mit der eingehenden Evaluation der Quellcodestrukturierung sowie der Betrachtung der Laufzeiteffizienz des Systems.

Der zweite Themenschwerpunkt beleuchtet die Konzeption und Entwicklung des *Peer-to-Peer*-Systems *Symstry*, das auf symbolischen Koordinaten operiert. Im Gegensatz zu *Symstry* operieren bestehende ortsbezogenen *Peer-to-Peer*-Systeme auf geometrischen Koordinaten und beschreiben die Welt mit Längen- und Breitengrad sowie einer optionalen Höhenangabe. Diese Verfahren sind jedoch nicht für alle Anwendungsszenarien ideal und zudem für den Menschen wenig intuitiv. Das auf Basis des Programmiermodells *Gears4Net* implementierte *Peer-to-Peer*-System verfügt über effiziente *Routing*- und *Geo-Cast*-Verfahren. Die Evaluation des Systems verdeutlicht, dass *Routing*-Nachrichten in *O(log n)* Schritten innerhalb des *Overlay*-Netzwerks zugestellt werden können. Zudem wird die Tiefe des für die Zustellung der *Geo-Cast*-Nachrichten aufgebauten Spannbaums von *O(log n)* nicht überschritten. Die Effizienz und Funktionalität des *Symstry Peer-to-Peer*-Systems wird abschließend durch die Bereitstellung der Beispielanwendung *Messenger* verdeutlicht.

Der dritte und letzte Themenschwerpunkt dieser Arbeit, das Kapitel *Peers@Play-Inspector*, betrachtet die Entwicklung einer Simulationsumgebung für *Peer-to-Peer*-basierte Systeme. Die Konzeption des *Peers@Play-Inspectors* sieht die Implementierung eines Werkzeugs vor, das eine *Peer-to-Peer*-Instanz nicht als monolithischen Block, sondern als eine aus verschiedenen Schichten aufgebaute Komponente betrachtet, die in verschiedenen Variationen kombiniert werden können. Zudem bestand die Forderung, ausschließlich produktiven Quellcode innerhalb der Simulationsumgebung zu verwenden und somit die Divergenz zwischen Produktiv- und Simulationsquellcode zu verhindern. Der *Peers@Play-Inspector*, der den aufgestellten Anforderungen genügt, bildet beispielsweise die Grundlage für die Evaluation des *Peer-to-Peer*-Systems *Symstry*.

Weiterer Forschungsbedarf - Ausblick

Die vorliegende Arbeit betrachtet bei der Beleuchtung des Entwicklungsprozesses massiv paralleler, verteilter Systeme viele unterschiedliche Fragestellungen aus verschiedensten Disziplinen der Informatik. Einzelne Detailfragen bleiben jedoch auch nach Abschluss der Arbeit offen beziehungsweise führen die während dieser Arbeit gewonnenen Erkenntnisse zu neuen Ansätzen, die die drei Forschungsthemen *Gears4Net*, *Symstry* und *Peers@Play-Inspector* weiter vorantreiben können.

Das *Gears4Net*-Programmiermodell erfordert die Einhaltung der zentralen Regel, dass zwei *Protocol*-Instanzen ausschließlich über unveränderte Nachrichten miteinander kommunizieren und dass innerhalb einer *Protocol*-Instanz keinerlei blockierende Systemaufrufe durchgeführt werden dürfen. Die Einhaltung dieser Regel kann in der vorliegenden Implementierung des *Gears4Net*-Modells nicht automatisiert überprüft werden. Eine zukünftige Erweiterung des *C#-Compilers* beziehungsweise die Bereitstellung eines *Pre-Compilers* könnte die Einhaltung der aufgestellten Regel forcieren und damit das *Gears4Net*-Programmiermodell einem größeren Zielpublikum öffnen.

Bei der Analyse des ortsbezogenen *Peer-to-Peer*-Systems *Symstry* stehen zwei Aspekte der Weiterentwicklung im Vordergrund. Der erste Aspekt betrachtet das Systemverhalten bei geographisch stark ungleich verteilten Knoten und die damit verbundenen Auswirkungen auf die verwendeten *Routing*-Mechanismen. Der zweite zu betrachtende Aspekt beinhaltet die Auswirkungen der Lokalitätseigenschaften des *Symstry*-Systems auf die Zusammenführung von partitionierten Netzwerksegmenten.

Den dritte und letzte Themenschwerpunkt dieser Arbeit bildet die Simulationsumgebung *Peers@Play-Inspector*. Die offene und erweiterbare Architektur des *Peers@Play-Inspector* ermöglicht die Einbindung von weiteren *Peer-to-Peer*-System *Plugins* und könnte darüber hinaus mit neuen Visualisierungs- und Managementfunktionen erweitert werden. Die vorgestellten Erweiterungskonzepte werden beispielsweise von zwei studentischen Projekten zur graphischen Darstellung der simulierten Netzwerktopologie beziehungsweise zum automatisierten *Software-Deployment* großer Testsysteme verwendet.

Abkürzungsverzeichnis

C#	C-Sharp
CCR	Concurrency and Coordination Runtime
CDS	Coordination Data Structures
CLR	Microsoft Common Language Runtime
DHT	Distributed Hash Table
DNS	Domain Name Systems
DRS	Distributed Runtime System
DSPT	Distributed data structure
GCP	Greatest Common Prefix
GPS	Global Positioning System
GUI	Graphical User Interface
ID	Identifizierer
JVM	Java Virtual Machine
LBM	Location Based Messengers
LDAP	Lightweight Directory Access Protocol
LINQ	Language Integrated Query
MSIL	Microsoft Intermediate Language
NAS	Netzwerkalterungssimulation
NAT	Network Address Translator
P@P	Peers@Play
P2P	Peer-to-Peer
PI	Protocol-Instanzen
PLINQ	Parallel Language Integrated Query
R	Zielgebiet/Region
RTE	Routing Tree Entry
S	Scheduler
SEDA	Staged Event-Driven Architecure
SM	State-Machine

SNAL	Secure Network Abstraction Layer
TCP	Transmission Control Protocol
TLD	Thread local Data
TLS	Thread local Storage
TPL	Task Parallel Library
UDP	User Datagram Protocol
VM	Virtual Memory
WS	Working Set

Literaturverzeichnis

[1] K. Aberer, L. O. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth, "The essence of p2p: A reference architecture for overlay networks," in *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*. IEEE Computer Society, 2005, pp. 11–20.

[2] S. Abramov, A. I. Adamovich, A. Inyukhin, A. Moskovsky, V. Roganov, E. Shevchuk, Y. Shevchuk, and A. Vodomerov, "Opents: An outline of dynamic parallelization approach," in *Parallel Computing Technologies, 8th International Conference (PaCT)*, 2005, pp. 303–312.

[3] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, pp. 1–72, 1998.

[5] A. L. Alan, A. Dearle, R. D. Bona, J. M. Farrow, F. Henskens, J. Rosenberg, and F. Vaughan, "A model for user-level memory management in a distributed, persistent environment," in *In Proceedings of the Seventeenth Annual Computer Science Conference, ACSC-17*, 1994, pp. 343–354.

[6] ANSI Standards for Information Technology, *Programming Languages C#, ISO/IEC 23270:2006(E) International Standard*, ISO-IEC, 2006.

[7] A. W. Appel, *Compiling with Continuations*. Cambridge University Press, 1992.

[8] T. Archer, *Inside C#*. Microsoft Press Deutschland, 2001.

[9] R. Bakshi, C. A. Knoblock, and S. Thakkar, "Exploiting online sources to accurately geocode addresses," in *GIS '04: Proceedings of the 12th annual ACM international workshop on Geographic information systems*. New York, NY, USA: ACM, 2004, pp. 194–203.

[10] S. A. Baset and H. G. Schulzrinne, "An analysis of the skype peer-to-peer internet telephony protocol," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, 2006, pp. 1–11.

[11] C. Becker, "System support for context-aware computing," Professorial Dissertation, June 2004, university of Stuttgart.

[12] N. Benton, L. Cardelli, and C. Fournet, "Modern Concurrency Abstractions for C#," in *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2002, pp. 415–440.

[13] N. Benton, L. Cardelli, and C. Fournet, "Modern concurrency abstractions for C#," *ACM Transactions on Programming Languages and System*, vol. 26, no. 5, pp. 769–804, September 2004.

[14] J. Beveridge and B. Wiener, *Multithreading applications in Win32: the complete guide to threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[15] G. Bierman, E. Meijer, and W. Schulte, "The essence of data access in comega: The power is in the dot," in *European Conference on Object-Oriented Programming (ECOOP) '02*, 2002.

[16] A. Birrell, J. Guttag, J. J. Horning, and R. Levin, "Synchronization primitives for a multiprocessor: A formal specification," *IGOPS Operating Systems Review*, vol. 21, no. 5, pp. 94–102, 1987.

[17] A. D. Birrell, "An introduction to programming with threads," Research Report 35, Digital Equipment Corporation Systems Research, Tech. Rep., 1989.

[18] R. Blum, *C# Network Programming*, S. Engelfried, Ed. SYBEX Inc., 2003.

[19] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[20] R. H. Campbell and A. N. Habermann, "The specification of process synchronization by path expressions," in *Operating Systems, Proceedings of an International Symposium*. London, UK: Springer-Verlag, 1974, pp. 89–102.

[21] R. H. Campbell and R. B. Kolstad, "Path expressions in pascal," in *ICSE '79: Proceedings of the 4th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1979, pp. 212–219.

[22] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "Scribe: a large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1489–1499, October 2002.

[23] V. G. Cerf and R. E. Icahn, "A protocol for packet network intercommunication," *IEEE Transactions on Communications*, vol. 22, pp. 637–648, 1974.

[24] M. K. Chandy, "Event-Driven Applications: Costs, Benefits and Design Approaches," in *Gartner Application Integration and Web Services Summit*, 2006.

[25] W. D. Clinger, "Foundations of actor semantics," Cambridge, MA, USA, Tech. Rep., 1981.

[26] D. Comer, *Internetworking with TCP/IP: principles, protocols, and architecture.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[27] M. Conrad and H.-J. Hof, "A generic, self-organizing, and distributed boostrap service for peer-to-peer networks," in *2nd International Workshop on Self-Organizing Systems (IWSOS)*, ser. Lecture Notes in Computer Science, vol. 4725/2007. Springer, 2007, pp. 59–72.

[28] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms.* McGraw-Hill Higher Education, 2001.

[29] G. D. O. Costa and A. Oliva, "Speeding up thread-local storage access in dynamic libraries," in *In Proceedings of the GCC Developer's Summit*, 2006, pp. 159–178.

[30] C. Cramer and T. Fuhrmann, "Bootstrapping Chord in Ad hoc Networks: Not Going Anywhere for a While," in *Proceedings of the 3rd IEEE International Workshop on Mobile Peer-to-Peer Computing*, Pisa, Italy, March 2006.

[31] C. Cramer, K. Kutzner, and T. Fuhrmann, "Bootstrapping Locality-Aware P2P Networks," in *Proceedings of the IEEE International Conference on Networks (ICON)*, Singapore, November 2004, pp. 357–361.

[32] Daan Leijen and Judd Hall, "Optimize managed code for multi-core machines," 2007. Online verfügbar: http://msdn.microsoft.com/en-us/magazine/cc163340.aspx Zuletzt aufgerufen am 09.02.2010.

[33] P. Darwen and X. Yao, "Every niching method has its niche: Fitness sharing and implicit sharing compared," in *In Proc. of Parallel Problem Solving from Nature (PPSN) IV - Lecture Notes in Computer Science 1141.* Springer-Verlag, 1996, pp. 398–407.

[34] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages: NATO Advanced Study Institute.* Academic Press, 1968, pp. 43–112.

[35] W. Ding, "Bootstrapping Chord over MANETs - All Roads Lead to Rome," in *IEEE Wireless Communications and Networking Conference (WCNC)*, Kowloon, China, March 11-15 2007, pp. 3501–3506.

[36] P. Druschel and A. Rowstron, "PAST: A persistent and anonymous store," in *The 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, May 2001.

[37] F. Dürr and K. Rothermel, "An overlay network for forwarding symbolically addressed geocast messages," in *Proceedings of the 15th International Conference on Computer Communications and Networks (ICCCN 2006)*, Arlington, VA, USA, Oct. 2006.

[38] ECMA International, "Standard ECMA-334 - C# Language Specification," 2006. Online verfügbar: http://www.ecma-international.org/publications/standards/Ecma-334.htm Zuletzt aufgerufen am 09.02.2010.

[39] ECMA International, "Standard ECMA-335 - Common Language Infrastructure (CLI)," 2006, zuletzt aufgerufen am 09.02.2010. Online verfügbar: http://www.ecma-international.org/publications/standards/Ecma-335.htm

[40] S. Y. Ghalsasi, "Critical success factors for event driven service oriented architecture," in *ICIS '09: Proceedings of the 2nd International Conference on Interaction Sciences*. New York, NY, USA: ACM, 2009, pp. 1441–1446.

[41] Google Inc., "Google maps," 2008. Online verfügbar: http://maps.google.de/ Zuletzt aufgerufen am 09.03.2010.

[42] Google Inc., "Places Directory," 2009. Online verfügbar: http://sites.google.com/site/placesdirectory/ Zuletzt aufgerufen am 09.02.2010.

[43] J. Goug, *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall PTR, 2002.

[44] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[45] Y. Gurevich, W. Schulte, and C. Wallace, "Investigating java concurrency using abstract state machines," in *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*. London, UK: Springer-Verlag, 2000, pp. 151–176.

[46] V. Guzev, "Parallel C#: The Usage of Chords and Higher-order Functions in the Design of Parallel Programming Languages," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2008, pp. 833–837.

[47] P. Haller, "Scala actors: A short tutorial," 2007. Online verfügbar: http://www.scala-lang.org/node/242/ Zuletzt aufgerufen am 09.02.2010.

[48] P. Haller and M. Odersky, "Actors that unify threads and events," *Coordination Models and Languages*, pp. 171–190, 2007.

[49] P. B. Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, vol. 1, no. 2, pp. 199–207, 1975.

[50] D. Heutelbeck and M. Hemmje, "A peer-to-peer data structure for dynamic location data," *Pervasive Computing and Communications, 2006. PerCom 2006. Fourth Annual IEEE International Conference on*, pp. 264–268, 2006.

[51] D. Heutelbeck, "Distributed Space Partitionin Trees and their Application in Mobile Computing," Ph.D. dissertation, Open University Hagen, Germany, May 2005.

[52] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[53] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, pp. 549–557, 1974.

[54] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 26, pp. 100–106, 1983.

[55] T. Imielinski and J. C. Navas, "GPS-based geographic addressing, routing, and resource discovery," *Communications of the ACM*, vol. 42, no. 4, pp. 86–92, 1999.

[56] Jabber Software Foundation, "Jabber protocols," 2006. Online verfügbar: http://www.jabber.org/protocol Zuletzt aufgerufen am 09.02.2010.

[57] K. Johns and T. Taylor, *Professional Microsoft Robotics Developer Studio.* Wiley Publishing Inc., 2008.

[58] Julio C. Navas and T. Imielinski, "Geocast – geographic addressing and routing," in *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '97)*, Budapest, Hungary, Sep. 1997, pp. 66–76.

[59] S. Klein, *Professional LINQ.* Birmingham, UK, UK: Wrox Press Ltd., 2008.

[60] M. Knoll, A. Wacker, G. Schiele, and T. Weis, "Decentralized bootstrapping in pervasive applications," in *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*, 2007.

[61] M. Knoll, A. Wacker, G. Schiele, and T. Weis, "Bootstrapping in peer-to-peer systems," in *14th IEEE International Conference on Parallel and Distributed Systems (ICPADS'08)*, Melbourne, Australia, December 2008.

[62] M. Knoll and T. Weis, "Optimizing Locality for Self-Organizing Context-based Systems," in *International Workshop on Self-Organizing Systems (IWSOS 2006), Passau, Germany*, 2006.

[63] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An Architecture for Global-scale Persistent Storage," in *Proceedings of ACM ASPLOS*. ACM, November 2000.

[64] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Reading, MA: Addison-Wesley, 1999.

[65] D. Lea, "The java.util.concurrent synchronizer framework," *Science of Computer Programming*, vol. 58, no. 3, pp. 293–309, 2008.

[66] O. Lehmann, M. Bauer, C. Becker, and D. Nicklas, "From home to world - supporting context-aware applications through world models," in *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*. Washington, DC, USA: IEEE Computer Society, 2004, p. 297.

[67] C. Maihöfer, "A Survey on Geocast Routing Protocols," *IEEE Communications Surveys and Tutorials*, vol. 6, no. 2, 2004.

[68] A. B. Mickel, J. F. Miner, K. Jensen, and N. Wirth, *Pascal user manual and report (4th ed.): ISO Pascal standard*. New York, NY, USA: Springer-Verlag New York, Inc., 1991.

[69] Microsoft Corporation, "Comparing the Timer Classes in the .NET Framework Class Library," 2004. Online verfügbar: http://msdn.microsoft.com/en-us/magazine/cc164015.aspx Zuletzt aufgerufen am 09.02.2010.

[70] Microsoft Corporation, "C# Language Specification Version 3.0," 2007. Online verfügbar: http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc Zuletzt aufgerufen am 09.02.2010.

[71] Microsoft Corporation, "Lambda-Ausdrücke (C#-Programmierhandbuch)," 2007. Online verfügbar: http://msdn.microsoft.com/de-de/library/bb397687.aspx Zuletzt aufgerufen am 09.02.2010.

[72] Microsoft Corporation, "Parallel LINQ - Running Queries On Multi-Core Processorse," 2007. Online verfügbar: http://msdn.microsoft.com/en-us/magazine/cc163329.aspx/ Zuletzt aufgerufen am 09.02.2010.

[73] Microsoft Corporation, "Bing maps," 2008. Online verfügbar: http://www.bing.com/maps/ Zuletzt aufgerufen am 09.03.2010.

[74] Microsoft Corporation, "Iron Python - Documentation," 2008. Online verfügbar: http://ironpython.net/documentation/ Zuletzt aufgerufen am 09.02.2010.

[75] Microsoft Corporation, "Xbox 360 Technical Specifications," 2008. Online verfügbar: http://support.xbox.com/support/en/us/xbox360/hardware/specifications/consolespecifications.aspx/ Zuletzt aufgerufen am 09.02.2010.

[76] Microsoft Corporation, "CCR Introduction," 2009. Online verfügbar: http://msdn.microsoft.com/en-us/library/bb648752.aspx Zuletzt aufgerufen am 09.02.2010.

[77] Microsoft Corporation, "Constraints on Type Parameters (C# Programming Guide)," 2009. Online verfügbar: http://msdn.microsoft.com/en-us/library/d5x73970%28VS.80%29.aspx Zuletzt aufgerufen am 09.02.2010.

[78] Microsoft Corporation, "Data Structures for Parallel Programming," 2010. Online verfügbar: http://msdn.microsoft.com/en-us/library/dd460718%28VS.100%29.aspx Zuletzt aufgerufen am 09.02.2010.

[79] Microsoft Corporation, "Introduction to PLINQ," 2010. Online verfügbar: http://msdn.microsoft.com/en-us/library/dd997425%28VS.100%29.aspx Zuletzt aufgerufen am 09.02.2010.

[80] Microsoft Corporation, "Parallel Programming in the .NET Framework," 2010. Online verfügbar: http://msdn.microsoft.com/en-us/library/dd460693%28VS.100%29.aspx Zuletzt aufgerufen am 09.02.2010.

[81] Microsoft Corporation, "Task parallel library," 2010. Online verfügbar: http://msdn.microsoft.com/en-us/library/dd460717%28VS.100%29.aspx Zuletzt aufgerufen am 09.02.2010.

[82] Microsoft Corporation, "Thread local storage," 2010. Online verfügbar: http://msdn.microsoft.com/en-us/library/ms686749%28VS.85%29.aspx/ Zuletzt aufgerufen am 09.02.2010.

[83] R. Milner, *Communication and concurrency*. Hertfordshire, UK: Prentice Hall International (UK) Ltd., 1995.

[84] R. Milner, *Communicating and mobile systems: the pi-calculus*. New York, NY, USA: Cambridge University Press, 1999.

[85] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

[86] M. Mintz, "MSN Messenger Protocol," 2004. Online verfügbar: http://www.hypothetic.org/docs/msn/index.php Zuletzt aufgerufen am 09.02.2010.

[87] A. Mislove, A. Post, A. Haeberlen, and P. Druschel, "Experiences in building and operating ePOST, a reliable peer-to-peer application," in *Proceedings of the 1st Conference of the European Professional Society for Systems (EuroSys'06)*, April 2006.

[88] B. Montrose, "A thread-local storage class for win32," *C/C++ Users Journal*, vol. 15, no. 11, pp. 49–53, 1997.

[89] A. Moskovsky, V. Roganov, and S. Abramov, "Parallelism Granules Aggregation with the T-System," in *Parallel Computing Technologies, 9th International Conference, PaCT*, 2007, pp. 293–302.

[90] A. Moskovsky, V. Roganov, S. Abramov, and A. Kuznetsov, "Variable Reassignment in the T++ Parallel Programming Language," in *Parallel Computing Technologies, 9th International Conference, PaCT*, 2007, pp. 579–588.

[91] C. Nagel, B. Evjen, J. Glynn, K. Watson, and M. Skinner, *Professional C# 2008*. Wiley Publishing Inc., 2008.

[92] J. C. Navas, "Geographic Routing in a Datagram Internetwork," Ph.D. dissertation, Rutgers University, Department of Computer Science, May 2001.

[93] Oracle - Sun Developer Network, "The Java Language Specification," 2005. Online verfügbar: http://java.sun.com/docs/books/jls/download/langspec-1.0.pdf Zuletzt aufgerufen am 09.02.2010.

[94] J. Ousterhout, "Why threads are a bad idea (for most purposes)," in *USENIX Technical Conference*, January 1996.

[95] C. Petzold, *Programming Windows*, 5th ed., Microsoft Press, Ed., 1999.

[96] J. Postel, "User Datagram Protocol," RFC 768 (Standard), Internet Engineering Task Force, Aug. 1980. Online verfügbar: http://www.ietf.org/rfc/rfc768.txt Zuletzt aufgerufen am 09.02.2010.

[97] J. Postel, "Transmission Control Protocol," RFC 793 (Standard), Internet Engineering Task Force, Sep. 1981. Online verfügbar: http://www.ietf.org/rfc/rfc793.txt Zuletzt aufgerufen am 09.02.2010.

[98] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The bittorrent p2p file-sharing system: Measurements and analysis," *Peer-to-Peer Systems IV*, pp. 205–216, 2005.

[99] PS3 Power, *PS3 Hardware Specs*, 2008, zuletzt aufgerufen am 09.02.2010. Online verfügbar: http://www.ps3power.com/ps3hardwarespecs.htm

[100] X. Qiu, G. C. Fox, and A. Ho, "Analysis of Concurrency and Coordination Runtime CCR and DSS for Parallel and Distributed Computing," 01/21/2007 2007. Online verfügbar: http://grids.ucs.indiana.edu/ptliupages/publications/CCRDSSanalysis_jan21-07.pdf

[101] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 161–172.

[102] J. Roth, "Semantic Geocast Using a Self-Organizing Infrastructure," in *Innovative Internet Community Systems (I2CS)*. Leipzig, Germany: Springer, Jun. 2003, pp. 216–228, LNCS 2877.

[103] K. Rothermel, M. Bauer, and C. Becker, *Digitale Weltmodelle - Grundlage kontextbezogener Systeme*, F. Mattern, Ed. Springer, 2003.

[104] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science*, vol. 2218, pp. 329–350, 2001.

[105] A. Rowstron and P. Druschel, "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 188–201, 2001.

[106] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*, ser. Lecture Notes in Computer Science, J. Crowcroft and M. Hofmann, Eds., vol. 2233, Nov. 2001, pp. 30–43.

[107] H. Sagan, *Space-Filling Curves*, J. Ewing, F. Gehring, and P. Halmos, Eds. New York, NY, USA: Springer-Verlag, 1994.

[108] D. Sangiorgi and D. Walker, *PI-Calculus: A Theory of Mobile Processes*. New York, NY, USA: Cambridge University Press, 2001.

[109] M. Saternus, M. Knoll, F. Dürr, and T. Weis, "Symstry: Ein P2P-System für Ortsbezogene Anwendungen," in *Proceedings of 15. ITG/GI - Fachtagung (KiVS 2007)*. VDE-Verlag, Februar 2007, Konferenz-Beitrag, pp. 99–104.

[110] M. Saternus, T. Weis, S. Holzapfel, and A. Wacker, "Gears4net - an asynchronous programming model," *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing - Workshops*, 2009.

[111] M. Saternus, T. Weis, M. Knoll, and F. Durr, "A middleware for context-aware applications and services based on messenger protocols," in *PERCOMW '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 467–471.

[112] H. Schildt, *C# 2.0: The Complete Reference*. Mcgraw-Hill Professional, 2nd ed., 2006.

[113] H. Schwichtenberg and F. Eller, *Programmierung mit der .NET-Klassenbibliothek*, 2nd ed. Addison-Wesley, 2003.

[114] J. Sermersheim, "Lightweight Directory Access Protocol (LDAP): The Protocol," RFC 4511 (Proposed Standard), Internet Engineering Task Force, Jun. 2006. Online verfügbar: http://www.ietf.org/rfc/rfc4511.txt Zuletzt aufgerufen am 09.02.2010.

[115] S. Singh and G. Chrysanthakopoulos, "An asynchronous messaging library for c#," *Synchronization and Concurrency in Object-Oriented-Languages (SCOOL)*, 2005.

[116] P. Srisuresh and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)," RFC 3022 (Informational), Jan. 2001. Online verfügbar: http://www.ietf.org/rfc/rfc3022.txt Zuletzt aufgerufen am 09.02.2010.

[117] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for internet applications," in *Proceedings of the 2001 ACM SIGCOMM Conference*. San Diego, California, United States: ACM, 2001, pp. 149–160.

[118] A. S. Tanenbaum, *Modern Operating System*. Pearson Education, Inc., 2008, vol. 3rd. ed.

[119] T. Titus, S. Gopikrishna, T. Redkar, S. Sivakumar, and F. C. Ferracchiati, *C# Threading Handbook*. Birmingham, UK: Wrox Press Ltd., 2003.

[120] S. Toub, "PATTERNS OF PARALLEL PROGRAMMING," 2010, zuletzt aufgerufen am 09.02.2010. Online verfügbar: http://www.microsoft.com/downloads/details.aspx

[121] University of Duisburg-Essen and University Mannheim, "peers@play homepage," 2008. Online verfügbar: http://www.peers-at-play.org/ Zuletzt aufgerufen am 09.02.2010.

[122] G. van Rossum and J. de Boer, "Linking a stub generator (AIL) to a prototyping language (Python)," in *EurOpen Conference*, 1991, pp. 229–247.

[123] A. Wacker, G. Schiele, S. Holzapfel, and T. Weis, "A NAT Traversal Mechanism for Peer-To-Peer Networks," in *P2P '08: Proceedings of the Eighth IEEE International Conference on Peer-to-Peer Computing (P2P'08)*. Aachen, Germany: IEEE, Sep. 2008, pp. 81–83.

[124] M. Wahl, T. Howes, and S. Kille, "Lightweight Directory Access Protocol (v3)," RFC 2251 (Proposed Standard), Internet Engineering Task Force, Dec. 1997. Online verfügbar: http://www.ietf.org/rfc/rfc2251.txt Zuletzt aufgerufen am 09.02.2010.

[125] K. Watson, *Beginning Visual C# 2005*. Wiley Publishing Inc., 2006.

[126] T. Weis, "Vorlesung Peer-to-Peer Systeme," 2009. Online verfügbar: http://vs.uni-due.de/index.php?view=article&id=98%3Ap2p-systeme&Itemid=67&option=com_content Zuletzt aufgerufen am 09.02.2010.

[127] M. Weiser, "The computer for the 21st century," *Mobile Computing and Communications Review (SIGMOBILE)*, vol. 3, no. 3, pp. 3–11, 1999.

[128] M. Welsh, "The Staged Event-Driven Architecture for Highly-Concurrent Server Applications," 2000. Online verfügbar: http://www.eecs.harvard.edu/~mdw/papers/quals-seda.pdf Zuletzt aufgerufen am 09.02.2010.

[129] M. Welsh, D. E. Culler, and E. A. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. Banff, Alberta, Canada: ACM Press, October 2001, pp. 230–243.